UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Gregor Jagnje

Simulacija črede ovc in psa goniča

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

Ljubljana, 2015

UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Gregor Jagnje

Simulacija črede ovc in psa goniča

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: doc. dr. Iztok Lebar Bajec

Ljubljana, 2015

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V diplomski nalogi preučite vir Strömbom, D., et al. (2014), Solving the shepherding problem: heuristics for herding autonomous, interacting agents, doi: 10.1098/rsif.2014.0719 in poustvarite predstavljeni algoritem črede ovc in psa goniča. V nadaljevanju najprej ponovite eksperimente, nato pa poizkusite delovanje algoritma izboljšati. Rezultate kritično komentirajte.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Gregor Jagnje, z vpisno številko 63100028 sem avtor diplomskega dela z naslovom:

Simulacija črede ovc in psa goniča

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr Iztoka Lebarja Bajca
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne _____ Podpis avtorja:_____

Kazalo

Kazalo slik

Seznam uporabljenih kratic

kratica	angleško	slovensko
OpenGL	Open Graphics Library	Odprta grafična knjižnjica
GCM	Global centre of mass	Globalno težišče
LCM	Local centre of mass	Lokalno težišče

Povzetek

Čredenje se velikokrat pojavlja pri računalniških simulacijah ter matematičnih modelih. Ti modeli poizkušajo poustvariti vedenje ptic, rib, insektov ter drugih črednih nagonov ostalih živali. Namen naloge je preučiti aktualno literaturo na področju simuliranja čred ovc, ter implementirati in nadgraditi model, ki simulira čredenje s pomočjo ovčarja. V diplomski nalogi so predstavljeni različni načini implementacije čredenja ter njihove prednosti in slabosti, prav tako je predstavljen razvoj in pregled delovanja črednega algoritma, ki temelji na različnih stanjih ovčarja. Algoritem poustvarja članek, ki je bil glavno vodilo za izdelavo simulacije [1], prav tako so poustvarjeni testi za uspešnost algoritma s članka, ter predlagane in implementirane možne izboljšave algoritma, katere rezultate nato primerjamo.

Ključne besede: simuliranje črede ovc, čredenje, k-najbližjih sosedov.

Abstract

Sheep herding often appears in computer simulations and mathematical models. These models try to replicate the behavior of bird flocks, fish schools, insect swarms and other flocking behaviors. The purpose of this thesis is to examine current literature in the field of sheep flocking behavior, then implementing and improving a model that simulates sheep herding with the help of a shepherd. The thesis shows different implementations of herding and their advantages and disadvantages, and also shows a development and examination of an algorithm based on shepherd states. The algorithm is recreated from an article that served as the main guideline for creating the simulation [1], the thesis also recreates success tests from the article, and suggests and implements a few possible improvements to the algorithm. The tests are then done again and compared.

Keywords: Sheep herding simulation, flocking, k-closest neighbours.

Poglavje 1 Uvod

Ovčarjenje je eno od najstarejših poklicov, saj se je začelo že približno pet tisoč let nazaj [2]. Ovce so ljudem uporabne zaradi mleka, mesa ter seveda njihove volne, zato se je ovčarjenje hitro razširilo čez celoten svet. Že v tistih časih so ovčarji skrbeli za ovce, jih držali v čredi ter peljati od pašnika do pašnika, njihov poklic se je tudi hitro ločil od kmetovanja, saj so ovčarji morali skrbeti za velike črede ovc naenkrat. Znanstveniki se s čredami ukvarjajo že dolgo časa, saj gre za zanimivo in uporabno področje raziskave tako pri matematiki kot računalniškem modeliranju [1], vendar področje še vedno ni popolnoma raziskano, saj ne obstaja en določen algoritem, ki bi natančno znal poustvariti obnašanje ovčarja ter ovc v čredi, prav tako je veliko neznanih dejavnikov, ki vplivajo na odločitve ovčarskih psov. Algoritmi, ki poizkušajo poustvariti vedenje čred, temeljijo na različnih metodah poustvarjanja čredenja kot pravila za ločitev, poravnavo ter kohezijo [1], [4], [8], agente s stanji, lahko uporabijo tudi A* algoritme za iskanje poti [11] in mnogo drugih rešitev. Čredni algoritmi se uporabljajo v industriji iger [4], v filmski industriji, robotiki, matematiki, računalniških simulacijah ter biologiji. Čredni algoritmi imajo tudi veliko možnih praktičnih uporab, saj bi lahko pomagali ali celo nadomestili ovčarje z roboti, uporabimo lahko majhne robote za pobiranje smeti, ali pa tudi usmerjamo ljudi kot pomoč za evakuacijo v slabo vidnih prostorih, prav tako so predlagani za pomoč pri čiščenju razlitja nafte, ter preprečevanja širjenja razlitja na širše območje [1], [3]. Diplomska naloga se osredotoča na model, kjer ovčar deluje na principu dveh stanj, vsako od teh stanj določa kam se mora ovčar postaviti da ovce v čredi pripelje do cilja.

V prvem delu diplomske naloge je predstavljenih nekaj modelov čredenja, ter kako delujejo, nato pa je predstavljen model iz članka, ki ga poizkušamo poustvariti [1]. V drugem delu diplomske naloge je predstavljena izdelava računalniškega modela simulacije nato pa v zadnjem delu razložimo nekaj hitrostnih izboljšav, ki so bile implementirane za hitrejše testiranje aplikacije ter prikažemo ugotovitve, analizo testov ter predlagamo možne izboljšave.

Poglavje 2 Uporabljena orodja in tehnologija

2.1 Visual Studio 2012

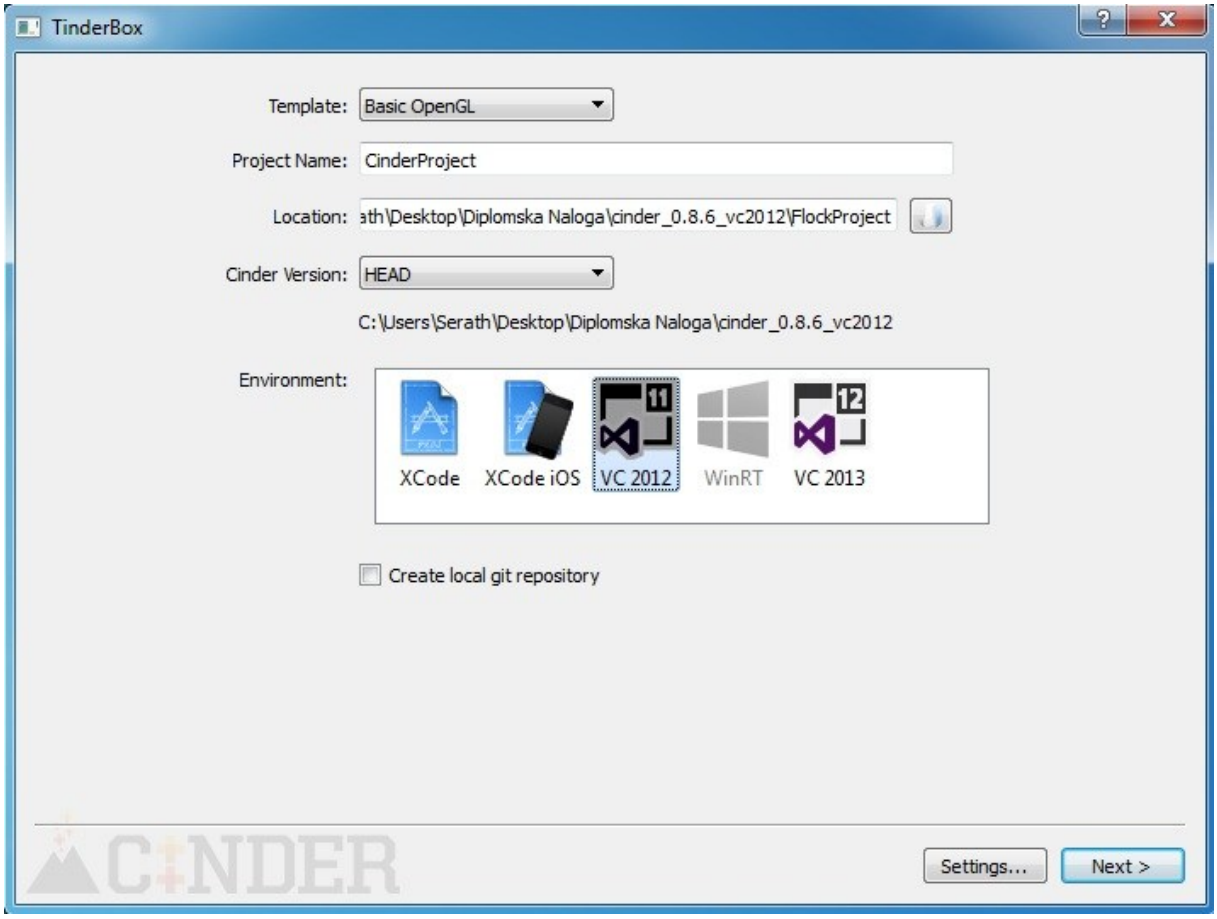
Microsoft je ustvaril razvojno okolje Visual Studio za razvoj aplikacij in grafičnih vmesnikov za operacijski sistem Windows. Uporablja se ga za razvoj spletnih strani, spletnih in mobilnih aplikacij ter medmrežnih storitev. Urejevalnik besedil v Visual Studiu podpira širok nabor programskih jezikov ter urejevalnikov. Znotraj Visual Studia se preprosto piše kodo programa, prav tako se to kodo preprosto zažene in prevede znotraj programa, kar nam omogoča hitro pisanje ter preverjanje kode [12].

2.2 Cinder

Cinder je odprtokodno intuitivno orodje za programiranje grafičnih, audio, video, spletnih ter drugih aplikacij. Orodje je narejeno tako, da aplikacije delujejo tako v Mac OS X, Windows ter mobilnih in drugih sistemih z isto kodo. Cinder se čim manj zanaša na tretje osebne knjižnjice, ter skuša čimbolje izkoristiti prednosti platforme na kateri programiramo. Za vizualizacijo uporablja OpenGL, ki omogoča upodabljanje 3D in 2D računalniške grafike. S pomočjo Cinderja v Visual Studiu lahko sprogramiramo vizualizacijo za simulacijo, ki nam služi za hitro preverjanje delovanja algoritmov [5].

2.2.1 Vzorec Cinder Projekta

Za izdelavo Cinder projekta potrebujemo prenesti Cinder za Visual Studio C++ 2012, v času izdelave diplomske naloge je bila najnovejša verzija Cinderja 0.8.6. K paketu je priložen Tinderbox, ki je preprost program za izdelavo novih projektov. V mapi *cinder_0.8.6_vc2012/tools/TinderBox-Win* poiščemo aplikacijo Tinderbox in jo zaženemo. Odpre se okno, v katerem lahko nastavimo kakšen projekt želimo ustvariti, kot prikazuje slika 2.1. Pod opcijo Template izberemo Basic OpenGL, napišemo ime projekta, ter izberemo kje želimo ustvariti ta projekt. Kot okolje moramo izbrati Visual Studio 2012, nato pa kliknemo next, ki nam odpre naslednje okno. V tem oknu lahko izberemo katera dodatna orodja želimo imeti ob začetku projekta, vendar jih za ta projekt ne potrebujemo, za to lahko kar ustvarimo projekt.



Slika 2.1: Okno za izdelavo Cinder projekta

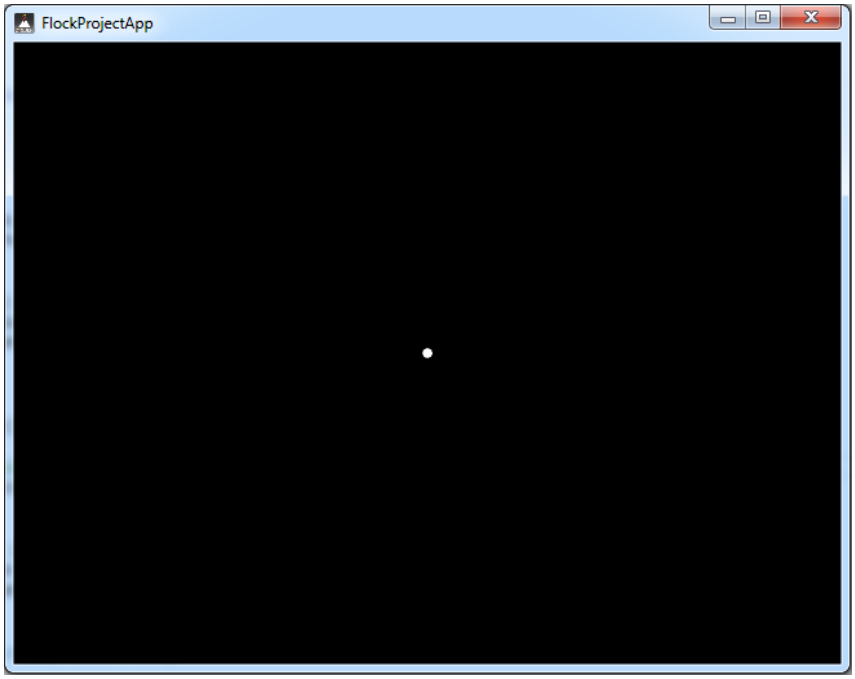
Cinder projekt se začne s tremi funkcijami, ki ločijo projekt na tri različne akcije. Projekt se začne z metodo `setup()` za nastavitev spremenljivk, kjer nastavimo začetne vrednosti spremenljivk, z metodo `update()` za posodobitev spremenljivk, kjer izračunamo nove vrednosti spremenljivk za vsak izris sličice v oknu aplikacije, ter metodo `draw()` za izris, kjer s posodobljenimi vrednostmi izrišemo trenutno stanje aplikacije oziroma simulacije. Poleg teh začetnih metod lahko dodamo tudi druge metode, kot so akcije na pritisk gumbov, pozicije računalniške miške ter ostalih dodatnih metod, vendar s temi osnovnimi tremi metodami lahko ustvarimo že večino aplikacij.

```
class FlockProjectApp : public AppBasic {
  public:
    void setup();
    void update();
    void draw();

    Vec3f               sheep, sheepVelocity;
    float               counter;
};

void FlockProjectApp::setup() {
    sheep = Vec3f(getWindowWidth() * 0.5f, getWindowHeight() * 0.5f, 0.0f);
    sheepVelocity = Vec3f(0.0f, 0.0f, 0.0f);
    counter = 0.0f;
}

void FlockProjectApp::update() {
    counter += 0.01f;
    sheepVelocity = Vec3f(sin(counter*M_PI), cos(counter*M_PI), 0.0f);
    sheep = sheep+sheepVelocity;
}

void FlockProjectApp::draw() {
    gl::clear( Color( 0, 0, 0 ) );
    gl::drawSphere(sheep, 4.0f, 8);
}

CINDER_APP_BASIC( FlockProjectApp, RendererGl )
```

Slika 2.2: Primer cinder aplikacije.

Znotraj teh osnovnih funkcij pišemo vso kodo, lahko pa tudi ustvarimo dodatne programske razrede, ki razporedijo kodo na več delov. Za začetek želimo ustvariti le preprosto aplikacijo (primer programa je na sliki 2.2), ki bo prikazala delovanje Cinderja, seveda je dobro da se držimo tematike diplomske naloge in zato ustvarimo ovco, ki se bo premikala po oknu. Za ustvarjanje aplikacije želimo pravilno uporabiti tri osnovne metode, zato moramo v `setup()` zapisati začetne vrednosti, v `update()` pisati računski in matematični del programa, v `draw()` pa postaviti vse metode, ki se tičejo izrisovanja. Ovco bomo postavili na sredino zaslona zato, da bo dobro vidna. Da ustvarimo ovco ji rabimo določiti vektor pozicije, za to v `setup()` metodi zapišemo tip podatka `Vec3f`, ki hrani tri koordinate, ki so potrebne za določitev točke v prostoru. Prvi dve koordinati postavimo tako, da poiščemo polovico širine ter višine okna v katerega izrisujemo, tretjo koordinato pa postavimo na nič, kjer bo tudi ostala, saj simuliramo kopenske živali. Naslednje kar rabimo za delovanje je hitrost ovce, vendar ko ovco ustvarimo se še ne premika, zato ji dodelimo ničelni vektor. V `draw()` metodi zapišemo, naj na poziciji ovce izriše kroglo. Privzeta barva je nastavljena na belo, torej nam je ni treba nastavljati, prav tako je bela barva povsem primerna za izris ovc, zato

nam je ni treba spreminjati. Če bi sedaj zagnali aplikacijo bi na sredini zaslona videli belo piko, ki stoji povsem pri miru, kot vidimo na sliki 2.3.



Slika 2.3: Sličica iz simulacije.

Ovci dodamo premikanje tako, da v `update()` metodo dodamo računanje nove pozicije ovce. Aplikaciji dodamo števnik, ki mu prištevamo neko vrednost za vsako izrisano sličico, ter dodamo vrednost hitrosti ovce. Nočemo, da bi nam ovca po nekaj sekundah zbežala izven vidnega polja, zato za njeno hitrost nastavimo krožno gibanje, kar pa dosežemo s preprosto uporabo funkcije sinusa in kosinusa. Funkciji določimo naj se s pomočjo števca sprehaja po sinusoidah, ter to hitrost prištejevamo poziciji ovce. Ob ponovnem zagonu aplikacije vidimo, da ovca kroži po zaslonu, ter tako izgubljeno išče svojega ovčarja.

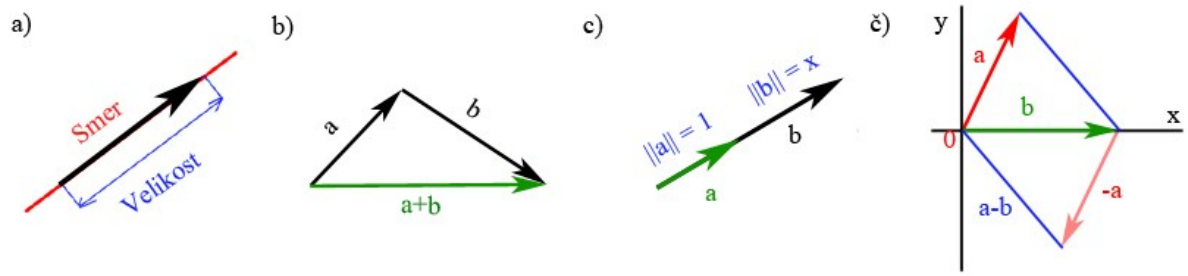
2.3 Photoshop CS5

Photoshop je program, ki ga uporabljajo fotografi, oblikovalci ter 3D umetniki za izboljšanje in manipulacijo fotografij ter ustvarjanje originalnih digitalnih umetniških del. Fotografi uporabljajo to orodje za popravljanje izpostavljenost leče pri slikanju, spremninjanje barv slikam, jim odrežejo robove, poravnajo slike ter jim izboljšajo kompozicijo. Uporablja se tudi za obnavljanje slik ter dodajanje popravkov slikam, popravljanje efekta rdečih oči od bliskavice ali odstranjevanje neželjenih lastnosti modelov. Prav tako se s tem programom

lahko združuje slike v kompozite, lahko dodajamo napise slikam, spreminjamo svetlobne efekte ter dodajamo filtre, ki spremenijo izgled slike [6].

2.4 Matematika za računanje z vektorji

Vektor je količina, ki ima poleg velikosti tudi smer, predstavljen je kot zaporedje treh števil, saj ima v 3D koordinatnem sistemu x, y ter z koordinato. V diplomski nalogi je uporabljen vektor na dva načina. Vektor je uporabljen le kot točka v koordinatnem sistemu, ki hrani lokacijo objekta, naj bo to ovca ali ovčar, prav tako uporabimo vektor kot količino, ki ima velikost in smer, saj ima vsak objekt tudi hitrost, ki določa kako hitro ter v katero smer se premika. Vektorsko računanje je glavni del diplomske naloge.



Slika 2.4: Od leve proti desni: Vektor, seštevanje, normalizacija ter odštevanje vektorjev

Glavne tri operacije računanja z vektorji, ki jih uporabljamo so seštevanje, normalizacija, ter odštevanje vektorjev, ki jih vidimo na sliki 2.4. Pri seštevanju dveh vektorjev seštejemo njuni koordinati ter tako dobimo novi vektor (2.1).

$$\bar{a} + \bar{b} = \begin{bmatrix} x_a \\ y_a \\ z_a \end{bmatrix} + \begin{bmatrix} x_b \\ y_b \\ z_b \end{bmatrix} = \begin{bmatrix} x_a + x_b \\ y_a + y_b \\ z_a + z_b \end{bmatrix} \tag{2.1}$$

V simulaciji uporabljamo to matematično operacijo pri premikanju ovc ter pastirja, saj če vektorju pozicije prištejemo usmerjen vektor hitrosti, bomo objekt na koordinatnem sistemu premaknili v smeri hitrosti, za njeno velikost. Seštevanje vektorjev prav tako uporabljamo pri seštevanju sil, ki vplivajo na ovce ter na ovčarja, kjer seštejemo vse vektorje sil za vsak objekt, ter tako dobimo končni usmerjeni vektor, ki določa v katero smer se bo objekt premaknil po seštevku vseh sil, ki vplivajo nanj. Velikost vektorja hitrosti določamo s pomočjo operacije normalizacije ter operacije množenja vektorja s skalarjem. Normalizacija vektorja pomeni, da vektorju obdržimo smer, vendar spremenimo njegovo velikost na enotsko

razdaljo. Pri normalizaciji najprej ugotovimo dolžino vektorja, to dosežemo z uporabo evklidske razdalje vektorja (2.2).

$$\bar{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{2.2}$$

$$\|\bar{a}\| = \sqrt{a_1^2 + a_2^2 + a_3^2}$$

Nato vzamemo vektor ki ga želimo normalizirati ter delimo vsako koordinato z razdaljo vektorja (2.3). Sedaj imamo vektor z enotsko razdaljo. Paziti rabimo le, da te operacije ne izvajamo na vektorjih dolžine nič, saj v teh primerih ne moremo koordinat deliti z razdaljo.

$$\bar{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{2.3}$$

$$\hat{a} = \begin{bmatrix} a_1/\|a\| \\ a_2/\|a\| \\ a_3/\|a\| \end{bmatrix}$$

$$\|\hat{a}\| = 1$$

Če hočemo sedaj temu vektorju določiti novo dolžino, ki v primeru hitrosti premikanja, pove kako hitro se bo premikal objekt, lahko uporabimo operacijo množenja vektorja s skalarjem. Vektor množimo s številko tako, da vsako koordinato vektorja množimo s to številko (2.4). Tako množenje spremeni le velikost vektorja, ne spremeni pa njegove smeri. S tem postopkom lahko zagotovimo, da se bodo agenti vedno premikali z določeno hitrostjo.

$$a * \bar{b} = a * \begin{bmatrix} x_b \\ y_b \\ z_b \end{bmatrix} = \begin{bmatrix} a * x_b \\ a * y_b \\ a * z_b \end{bmatrix} \tag{2.4}$$

Pri večini računskih operacij z vektorji potrebujemo ugotoviti razdalje med dvema agentoma, ter ugotoviti tudi smer v kateri je ta agent. Če odštejemo vektor agenta, ki ga gledamo, od agenta katerega hočemo ugotoviti smer in razdaljo, bomo dobili vektor, ki predstavlja v kateri smeri je drugi agent (2.5). V tri-dimenzionalnem prostoru je to vektor, ki se začne pri prvem agentu ter konča pri drugemu.

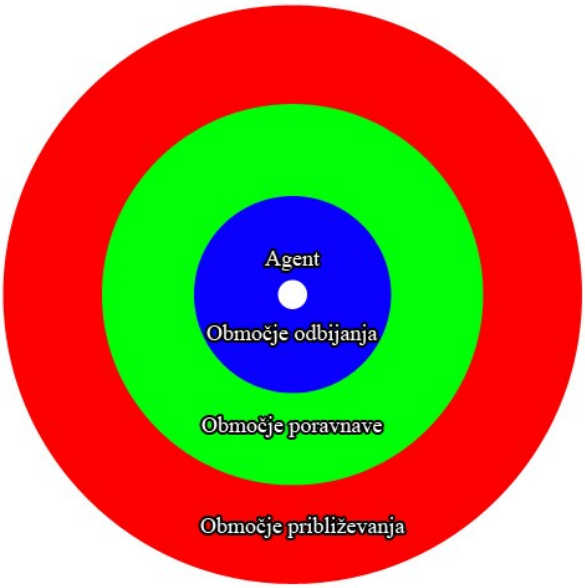
$$\bar{a} - \bar{b} = \begin{bmatrix} x_a \\ y_a \\ z_a \end{bmatrix} - \begin{bmatrix} x_b \\ y_b \\ z_b \end{bmatrix} = \begin{bmatrix} x_a - x_b \\ y_a - y_b \\ z_a - z_b \end{bmatrix} \tag{2.5}$$

Iz smernega vektorja lahko izračunamo razdaljo med dvema agentoma. To je potrebno tako pri preprečevanju trkov med agenti, kot pri preverjanju ali je ovčar že dovolj blizu ovc, da ga ovce lahko vidijo ter začnejo bežati od njega. Za izračun dolžine smernega vektorja med agentoma potrebujemo le izračunati evklidsko razdaljo vektorja, kar pomeni da kvadriramo razliko vsake koordinate, nato pa seštevek teh razlik korenimo (2.6).

$$\|\bar{a} - \bar{b}\| = \sqrt{\left(\begin{bmatrix} x_a \\ y_a \\ z_a \end{bmatrix} - \begin{bmatrix} x_b \\ y_b \\ z_b \end{bmatrix}\right)^2} = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2 + (z_a - z_b)^2} \tag{2.6}$$

2.5 Simulacije čredenja

Pri simuliranju jat ptic in rib se velikokrat uporabi model, ki temelji na treh območjih okoli vsakega agenta. Ta tri območja so poimenovana po njiovih lastnosti, namreč območje odbijanja, območje poravnave ter območje približevanja. Primer teh treh območij okoli agenta prikazuje slika 2.5.



Slika 2.5: Območja odbijanja poravnave ter približevanja okoli agenta.

Vsak agent se poizkuša pridružiti jati in ostati član te jate. Območje odbijanja je najmanjše območje okoli agenta, in poskrbi, da agenti med sabo nimajo trkov, to je ponavadi narejeno tako, da agent, ki zazna druge agente znotraj tega območja, dobi silo v nasprotni smeri od agentov, in se tako odmakne od agentov, ki so mu preblizu. Območje poravnave je območje znotraj katerega poizkuša agent imeti vse ostale agente. Ko so drugi agenti v njem se jim poskuša prilagoditi, ter potovati z enako hitrostjo ter v enako smer kot ostali. Če agent zazna drugega agenta znotraj območja približevanja pridobi silo proti temu agentu ter se mu tako poskuša približati. Velikosti teh območij prilagodimo glede na vrsto agentov ter na živali, ki jih poskušamo simulirati, prav tako prilagodimo velikosti sil, ki vplivajo na agente. Model s temi tremi območji se imenuje Boidov model [4], [8]. Temu modelu se lahko doda tudi druge lastnosti, kot učinki strahu [13], ki lahko pospešijo vsakega agenta, ter mu tako omogočijo beg pred plenilcem. Pri pticah, ki se selijo proti severu ali jugu, lahko dodamo ciljno usmerjenost, saj se lahko orientirajo preko sonca ali elektromagnetnih čutil, lahko tudi iščejo vizualne orientacijske znake, ter tako prispejo na lokacije tudi na drugih straneh sveta [9].

Dodamo lahko tudi dodatne sile k območju poravnave, kjer je v jati določena vodja, ki ima vpliv na premikanje celotne jate [14].

Pri simulacijah čred ovc lahko vzamemo Boidov model in ga omejimo na 2D polje. Pri tem ne rabimo imeti nobene kontrole nad čredo, ta se pojavi kot posledica težnje vsakega agenta, ki hoče ostati čimbližje ostalih agentov. Problem pri Boidovem modelu se pojavi, ko so si agenti predaleč narazen, in v neomejenem prostoru se lahko zgodi, da se agenti nikoli ne najdejo, ter se tako ne povežejo v čredo. To lahko rešimo tako, da agentom dodamo obnašanje, ki jim pomaga ustvariti čredo, ko se jim približa plenilec.

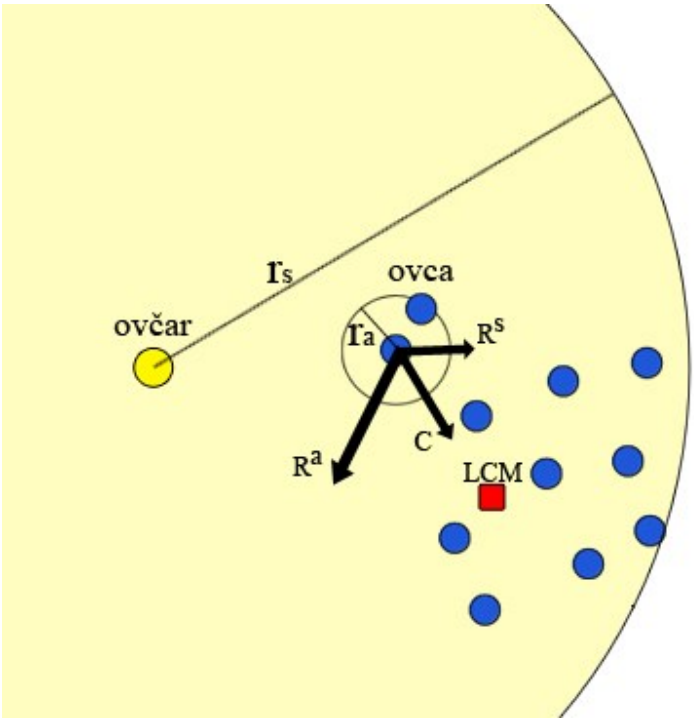
2.6 Razlaga uporabljenega modela

Pri simulacijah čredenja ovc poizkušamo poustvariti obnašanje ovc ter ovčarja. Ugotoviti hočemo kaj vse vpliva na ovce, kako se odločajo ter na kakšen način se združujejo v čredo, prav tako hočemo ugotoviti kako razmišlja pes ovčar, kako spremlja ovce ter na kakšen način poskrbi, da obdrži ogromne črede ovc v povezani gruči. Obstaja že veliko modelov, ki poustvarjajo čredenja, ki uspešno držijo ptice ali ribe v jati, insekte v rojih ter ostale živali v čredah. Velikokrat se ti modeli osredotočajo na koncept privlačne sile, poravnave ter odbijanja. Te tri sile, vsaka deluje v svojem območju okoli agenta, lahko uspešno poustvari večino vrst čredenja. V primeru ovc preprosto uporabiti tak model ne bo dobro poustvarilo delovanja njihovega črednega nagona, saj se ovce v naravi prosto pasejo dokler v njihovo bližino ne pride plenilec. Ovce so zelo družabna bitja, ko se pasejo hočejo še vedno biti dovolj blizu ostalim ovcam, da jih lahko vidijo. Prav tako nimajo dobro razvite obrambe pred plenilci, imajo pa dobro razvita čutila in se zanašajo na svoj čredni nagon da se obranijo poškodbam ali celo smrtnim žrtvam [10]. Ob približujoči nevarnosti se ovce hitro združijo v tesno čredo tako, da se premikajo proti sredini črede, saj je za ovco, ki ima druge ovce med sabo in plenilcem lažje preživeti, kot pa za ovco, ki je sama na paši.

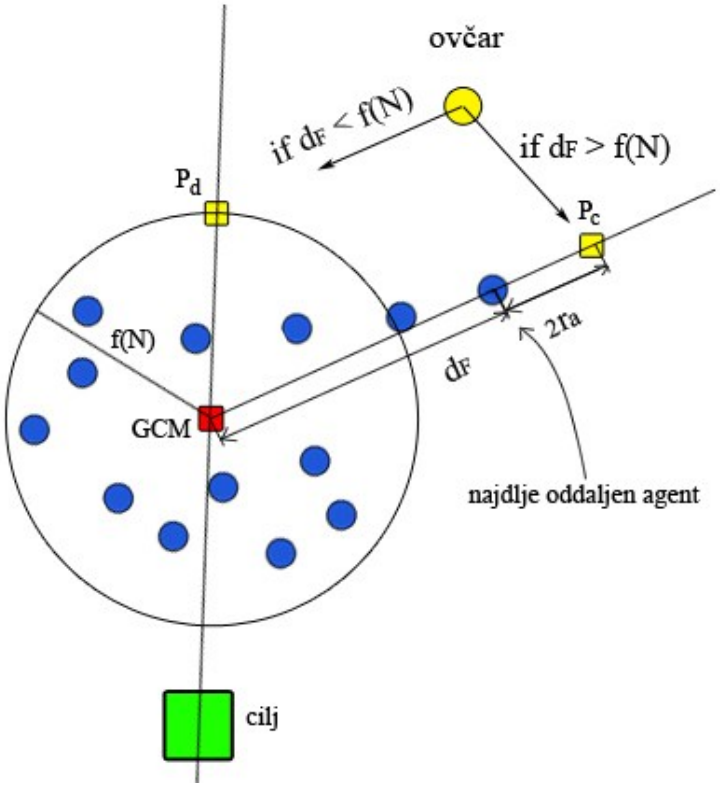
2.6.1 Ovca

Za delovanje algoritma potrebujemo ovčarju podati ovco ali čredo ovc, ki jih bo skušal usmeriti proti cilju. Ovce se obnašajo drugače, če je v bližini ovčar, kot takrat ko ga ni. Ob odsotnosti ovčarja se ovce prosto pasejo, v tem stanju bo ovca večino časa stala pri miru. V tem stanju ovce jedo travo, in se počasi premikajo ter tako vedno iščejo boljšo zaplato trave na kateri se pasejo. To v simulaciji poustvarimo tako, da se ovca občasno premakne v naključno smer. Ko se ovčar dovolj približa ovcam jih prestraši, v tem stanju ovce poiščejo k svojih najbližjih sosedov ter se približujejo centru mase teh ovc. Ko je ovčar dovolj blizu ovc, bodo ovce bežale od njega. V obeh stanjih se ovce odbijajo tudi med sabo, ter tako

preprečujejo trke. Na sliki 2.6 vidimo ilustracijo delovanja sil na ovci znotraj območja ovčarja. Območje ovčarja je označeno z rumeno barvo. Slika prikazuje smer sile R^s, ki potiska ovco stran od ovčarja, sile C, ki vleče ovco proti LCM, lokalnemu težišču njenih k najbližjih sosedov, ter silo R^a, ki potiska ovco stran od ovce znotraj njenega območja odbijanja velikosti r_a.



Slika 2.6: Sile, ki vplivajo na posamezne ovce.

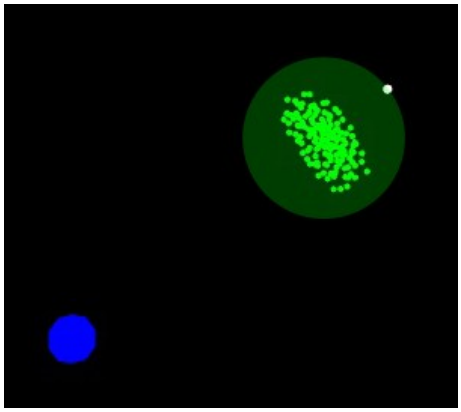


Slika 2.7: Sile, ki vplivajo na ovčarja.

2.6.2 Ovčar

Ovce se v naravi pasejo po travnikih in se po njih počasi sprehajajo med tem ko prežijo na nevarnosti. Takoj ko ovce zaznajo plenilca se odzovejo tako, da ustvarijo čredo. Za lažje vodenje ovc izkoristimo njihov čredni nagon tako, da uporabimo človeka ali zdresiranega psa, ki se približa čredi ter tako pripravi ovce, da se združijo v čredo in postanejo bolj obladljive. Slika 2.7 prikazuje kako ovčar v modelu obvladuje ovce s pomočjo pravilne postavitvije ter naravnega odziva ovc, da se odmikajo stran od plenilca. Ovčar lahko usmeri ovce kamor jih želi tako, da se postavi za ovce in jih vodi proti cilju. V simulaciji skušamo to vedenje ovc poustvariti, zato ustvarimo pastirskega psa, ki se premika hitreje od ovc in jih tako učinkovito vodi do cilja. Če se ovčar preveč približa ovcam se ustavi ter počaka, da se od njega ovce spet odmaknejo dovolj daleč stran, s tem preprečuje, da bi se preveč približal sredini črede, saj se v tem primeru ovce razcepijo. Ko so vse ovce v tesni čredi in je ovčar postavljen na točki P_d, zadaj za ovcami, relativno glede na cilj, pravimo, da je ovčar v gonilnem stanju. V tem stanju ovčar potiska ovce proti cilju, vendar se pogosto zgodi, da kakšna ovca uide iz črede. V tem primeru mora ovčar prenehati voditi čredo proti cilju, ter poiskati ovco, ter ji pomagati, da se pridruži nazaj čredi. Ko ovčar zapusti gonilno stanje, preide v zbiralno stanje. To je stanje kjer

se ovčar premakne na točko P_c, takoj za najdlje oddaljeno ovco, relativno glede na sredino črede in jo tako vodi nazaj proti čredi. Ker se ovčar pomika hitreje od ovc, lahko vedno prehiti ovco, ter se postavi za njo. Za premikanje med temi dvemi stanji preverjamo, če so vse ovce oddaljene manj od razdalje f(N) od središča mase vseh ovc. Ta razdalja je odvisna od števila ovc ter je prirejena tako, da ne zahtevamo, da so ovce znotraj popolnega kroga od središča, ampak jim dovolimo več prostora in dopustimo, je oblika črede lahko tudi elipsasta. V diplomskem delu sem imel težave pri nastavitviji f(N) na enako vrednost, kot je napisana v članku [1], saj je bila velikost območja okoli GCM prevelika pri f(N) vrednosti $r_a*N^{2/3}$. Zaradi tega je vrednost f(N) nastavljena na $r_a*\sqrt{N}$, ki še vedno dovoljuje elipsasto obliko črede. Na sliki 2.8 vidimo primer velikosti f(N) z enačbo $r_a*\sqrt{N}$ pri največjem številu ovc, ki jih uporabimo za testiranje.



Slika 2.8: Velikost območja f(N) pri vrednosti N = 150.

Pri opazovanju ovčarjevih premikov je pogosto omenjeno gibanje levo in desno za ovcami, saj to omogoča lažje čredenje ovc in jih dlje časa drži v tesni skupini. Velikokrat se v drugih simulacijah pojavi, da je tako vedenje kodirano v obnašanje ovčarja, vendar pri predlaganem modelu, do takega vedenja pride zaradi prehodov med gonilnim in zbiralnim stanjem.

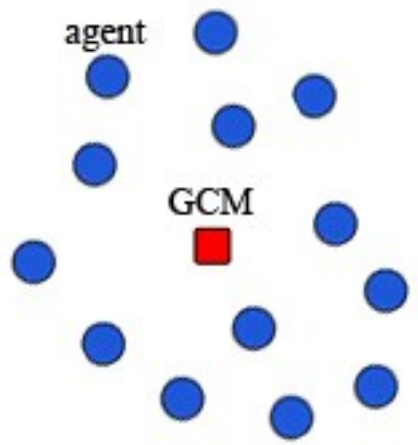
Poglavje 3 Izdelava simulacije čredenja ovc

Za lažje programiranje ločimo program na več delov ter ustvarimo dodatne razrede tako, da postane program bolj pregleden ter lažji za spreminjati. Ustvarimo razred upravljavca agentov, ki bo hranil seznam vseh ovc ter ovčarja, vseboval bo tudi večino računskega dela naloge. Ustvarimo tudi razred za ovco ter razred za ovčarja, ki bosta vsebovala vektorje hitrosti ter vektorje pozicije za vsako ovco in ovčarja. V `update()`in `draw()` metodah sedaj kličemo `update()` in `draw()` metodo od upravljavca agentov, saj hrani vse objekte, in se sprehodimo po seznamu ovc, ter kličemo `update()` in `draw()` metode za vsako ovco. Enako naredimo tudi za ovčarja čeprav vsebuje le en objekt. V razredu ovčarja ter v razredu ovce v `update()` metodi prištevamo vektor hitrosti k vektorju pozicije, v `draw()` metodah teh dveh razredov pa nastavimo barvo s katero bomo pobarvali ploskve pri izrisovanju ter izrišemo kroglo na sredini vektorju pozicije agenta.

3.1 Računanje GCM

Takoj na začetku računske zanke v razredu upravljalca agentov izračunamo globalni center mase ovc, to naredimo tako da seštejemo pozicije vseh ovc, ter na koncu dobljeni vektor delimo s številom ovc. Za namen simulacije so vse ovce enako velike ter težke, zato pri taki operaciji dobimo GCM ali globalni center mase ovc. Postopek prikazuje enačba (3.1), pri kateri je *posA* pozicija agenta, ter *posGCM* pozicija globalnega centra mase.. Lokacija GCM je točka, ki predstavlja sredino črede, kot prikazuje slika 3.1. GCM računamo na začetku, ker naslednje operacije že potrebujejo izračunan GCM za določitev točke pozicije gonilnega ali zbiralnega stanja za ovčarja, ter za določitev v katerem stanju je ovčar.

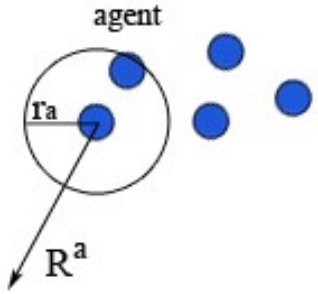
$$posGCM = \frac{\sum_{j=1}^{N} posA_j}{N} \tag{3.1}$$



Slika 3.1: Prikaz lokacije GCM v čredi agentov.

3.2 Preprečevanje trkov med agenti

Po izračunu GCM preverimo razdalje agentov med sabo in preverimo ali so znotraj območij odbijanja. Vsakemu agentu, ki ima agente preblizu, prištejemo vektor hitrosti R^a usmerjen stran od teh agentov, kot vidimo na sliki 3.2. Ta vektor hitrosti dobimo tako, da vektor pozicije drugega agenta odštejemo prvemu in dobimo vektor obrnjen stran od agenta, ki je preblizu kot prikazuje enačba (3.2), kjer je *posA* pozicija, *Va* pa hitrost prvega agenta ter *PosB* pozicija drugega agenta. Dobljeni vektor normaliziramo, da dobimo enotski vektor, nato ga pa množimo z odbojno silo med agenti ρ_a. Ta vektor prištejemo hitrosti agenta ter tako vplivamo na njegovo končno smer, in ga usmerimo stran od drugih agentov.



Slika 3.2: Odbojna sila med agenti.

$$\hat{R}_a = \frac{posB - posA}{\|posB - posA\|} \tag{3.2}$$

$$Va = Va + \hat{R}_a * \rho_a$$

3.3 Računanje k najbližjih sosedov

Naslednje razdalje, ki jih računamo so razdalje med agenti in ovčarjem. Za vsako ovco preverimo ali je znotraj območja vpliva ovčarja. Če ugotovimo, da je ovca znotraj vpliva ovčarja ji vključimo čredni nagon. Agentu poiščemo k njegovih najbližjih sosedov ter izračunamo njihov LCM, lokalni center mase. Hitrosti agenta nato prištejemo normaliziran vektor usmerjen od njega proti LCM, pomnožen s privlačno silo med ovcami c. Slika 3.3 prikazuje primer iskanja lokalnega centra mase pri vrednosti k = 3, na sliki so z zeleno označeni agenti, ki so najbližje trenutnemu agentu, ki ga gledamo. K najbližjih sosedov lahko poiščemo na veliko načinov, en najlažjih načinov kako pridemo do njega je tako, da pripišemo vsakemu agentu v seznamu oddaljenost od agenta, ki ga gledamo. Agenta, ki ga gledamo, ne smemo šteti kot soseda, zato mu pripišemo največjo razdaljo v seznamu. To razdaljo določimo tako, da agentu pripišemo največjo možno razdaljo, ki jo podpira float. Seznam nato sortiramo po dolžinah naraščujoče ter se sprehodimo po seznamu. Iz seznama vzamemo prvih k elementov, s katerimo lahko izračunamo LCM najbližjih sosedov. LCM izračunamo tako, seštejemo pozicije prvih k agentov v sortiranemu seznamu ter seštevek delimo s k. Agentu nato prištejemo hitrost proti LCM, ki ga dobimo tako, da pozicijo posA agenta odštejemo od pozicije LCM, ter dobljeni vektor normaliziramo, kot kaže enačba (3.3). Agentu prav tako dodamo tudi odbojno silo od ovčarja ρ_s, saj smo že znotraj zanke od ovčarja in že imamo izračunan vektor razdalje med agentom in ovčarjem. Odbojno silo od ovčarja določimo tako, da pozicijo ovčarja odštejemo od pozicije agenta, ter množimo normaliziran vektor s silo odboja ρ_s, kot prikazuje enačba (3.4).

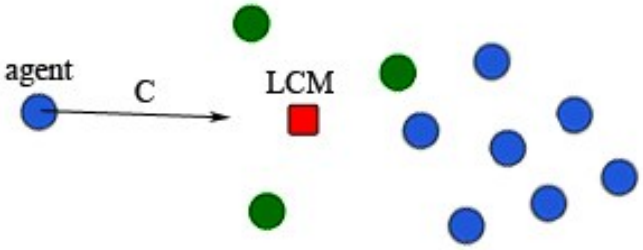
$$posLCM = \frac{\sum_{j=1}^{k} posA_j}{k} \tag{3.3}$$

$$\hat{C}_t = \|posA - posLCM\|$$

$$Va = Va + \widehat{C_t} * c$$

$$\hat{R}^s = \|posS - posA\| \tag{3.4}$$

$$Va = Va + \hat{R}^s * \rho_s$$



Slika 3.3: Prikaz iskanja LCM.

3.4 Naključno premikanje

Če ugotovimo, da je agent od ovčarja oddaljen dlje kot r_s se agent prosto pase. Agentu, ki se pase dodamo naključno gibanje, ki se aktivira z verjetnostjo p na vsako izračunano sličico. Ko se agent prosto pase bo vsako sličico dobil naključno število med 1 in 100, če bo ta številka manjša ali enaka od 5 generiramo vektor usmerjen v naključno smer ter ga normaliziramo. Naključni vektor $\in$ nato pomnožimo z velikostjo naključne sile e ter ga prištejemo vektorju hitrosti Va agenta, kot vidimo v enačbi (3.5). Ovčar ima prav tako kot ovce vgrajeno naključno premikanje z enako verjetnostjo kot ovce, vendar za razliko od ovc je to naključno premikanje vedno prisotno, simulacija pa ga vsebuje zaradi določenih primerov, kjer bi lahko ovčar prišel do mrtve točke v programu. Primer take mrtve točke bi bil, če bi imeli le eno ovco (N enak 1) in, če bi se ovčar pojavil natančno med ovco ter ciljem. V tem primeru bi ovčar imel gonilno točko takoj za agentom, njegova sila odbijanja bi agenta odmikala od cilja, ovčar pa ne more priti mimo agenta, saj se ustavi takoj ko pride $3r_a$ blizu agenta. Agent in ovčar bi se tako do konca izteka simulacije odmikala stran od cilja.

$$Va = Va + e * \in \tag{3.5}$$

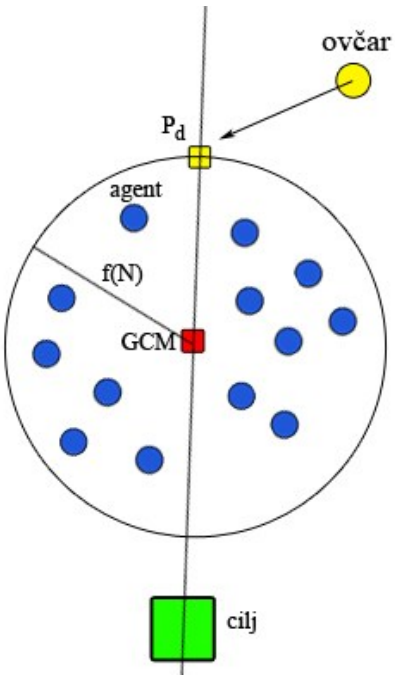
3.5 Gonilno stanje ovčarja

Za vsako ovco tudi preverimo oddaljenost od GCM, ter iščemo ovco, ki je najdlje oddaljena od GCM. Da preverjamo ali so ovce dovolj skupaj, da tvorijo čredo, določimo razdaljo f(N), ki je odvisna od števila ovc v čredi. Dokler so vse ovce znotraj f(N) bo ovčar v gonilnem stanju, če najdemo ovco, ki je oddaljena več kot f(N) od GCM bo ovčar šel v zbiralno stanje.

Razdaljo f(N) določimo tako, da vzamemo območje odbijanja med agenti ter pomnožimo s korenom števila ovc kot prikazuje enačba (3.6). V gonilnem stanju vzamemo ciljno lokacijo *posWin* ter jo odštejemo od GCM. Tako dobimo vektor, ki povezuje središče črede ovc s ciljno lokacijo, ta vektor normaliziramo ter množimo s f(N) ter prištejemo GCM-ju ovc in tako dobimo točko takoj za čredo P_d, kot prikazuje enačba (3.7). Če ovčarja postavimo na to gonilno točko se bodo ovce premikale stran od ovčarja, ter tako premikale GCM proti cilju, premikanju GCM-ja bo sledil tudi ovčar ter tako potiskal čredo proti cilju. Da ovčarja usmerimo proti gonilni lokaciji vzamemo njegov vektor pozicije ter ga odštejemo od gonilne lokacije, tako dobimo vektor proti gonilni lokaciji. Ta vektor normaliziramo ter ga prištejemo vektorju hitrosti ovčarja. Slika 3.4 prikazuje primer kjer je ovčar v gonilnem stanju.

$$f(N) = r_a * \sqrt{N} \tag{3.6}$$

$$P_d = posGCM + \|posWin - posGCM\| * r_a * f(N) \tag{3.7}$$



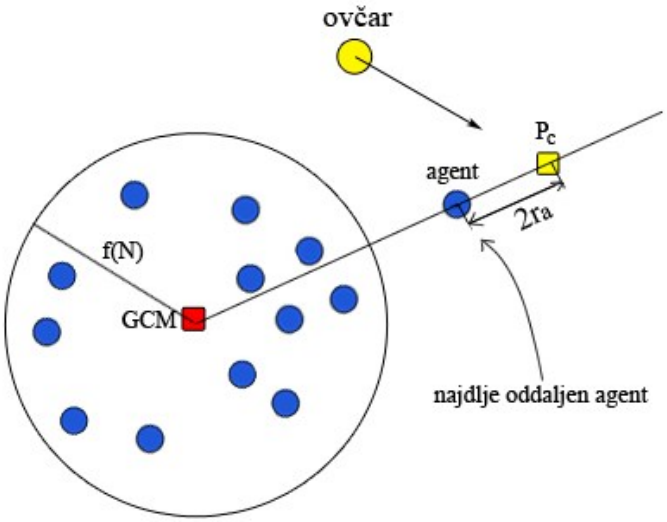
Slika 3.4: Prikaz delovanja ovčarja v gonilnem stanju.

3.6 Zbiralno stanje ovčarja

Če je ovčar v zbiralnem stanju pomeni, da je ena ali več ovc zunaj f(N) okoli GCM, kar pomeni, da rabimo izgubljene agente pripeljati nazaj v čredo. To naredimo tako, da odštejemo vektor pozicije GCM od pozicije agenta, ki je najdlje oddaljen od GCM, ter

dobljeni smerni vektor normaliziramo. Da dobimo točko takoj za tem agentom, prištejemo vektorju pozicije od agenta normalizirani vektor zmnožen z dvakratno razdaljo r_a. Dobili smo točko P_c, ki je takoj za agentom, relativno na GCM, kot prikazuje enačba (3.8), kjer je spremenljivka $A_{furthest}$ pozicija najdlje oddaljene ovce. Ovčarja v zbiralnem stanju prav tako prikazuje slika 3.5, kjer je tudi prikazana točka P_c. Če postavimo ovčarja na to točko se bo agent začel odmikati stran od ovčarja ter proti sredini črede. Da postavimo ovčarja na to točko vzamemo vektor pozicije od ovčarja ter jo odštejemo od dobljene točke, tako dobimo smerni vektor proti njej in ta vektor lahko prištejemo vektorju hitrosti ovčarja.

$$P_c = A_{furthest} + \|PosGCM - A_{furthest}\| * 2r_a \tag{3.8}$$



Slika 3.5: Prikaz delovanja ovčarja v zbiralnem stanju.

Po zahtevah napisanih v članku, ki ga poustvarjamo je potrebno zagotoviti, da se ovce premikajo s hitrostjo δ, ovčar pa s hitrostjo δ_s oziroma, da se ovce premikajo za en meter na časovni korak, med tem ko se ovčar premika za 1,5 m na časovni korak. To zagotovimo tako, da normaliziramo vektorje hitrosti, ki smo jih dobili s seštevanjem sil, tako pri ovcah kot pri ovčarju, ter jih pomnožimo z δ za ovce ter δ_s za ovčarja, kot prikazuje enačba (3.9). Dobljeni vektor hitrosti na koncu v `update()` metodi od ovc ter `update()` metodi od ovčarja prištejemo k poziciji, ter tako premaknemo vektorje za to dolžino na novo pozicijo.

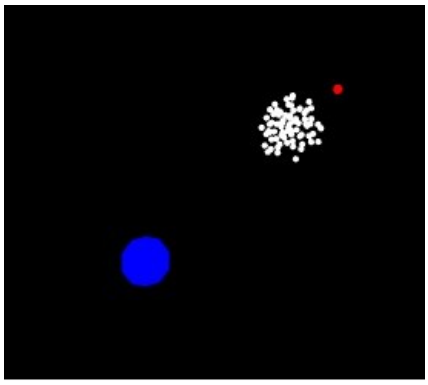
$$Va = \|Va\| * \delta \tag{3.9}$$

$$Vs = \|Vs\| * \delta_s$$

Oznaka parametra	**Opis**	**Vrednost v simulaciji**
L	Velikost stranice kvadrata	150 m
Parametri agentov		
N	Število agentov	1 - 150
k	Številno najbližjih sosedov	1 - N
$\boldsymbol{r_s}$	Območje zaznave ovčarja	65 m
$\boldsymbol{r_a}$	Območje odbijanja med agenti	2 m
$\boldsymbol{\rho_a}$	Sila odbijanja med agenti	2
$\boldsymbol{c}$	Privlačna sila med k-najbližjimi sosedi	1.05
$\boldsymbol{\rho_s}$	Sila odbijanja od ovčarja	1
h	Vztrajnost	0.5
e	Naključna sila	0.3
$\boldsymbol{\delta}$	Velikost premika na časovni korak	1 m
p	Možnost premika z naključno silo	0.05
f(N)	Največja velikost združene črede	$r_a \mathrm{N}^{1/2}$
Parametri ovčarja		
$\boldsymbol{\delta_s}$	Velikost premika na časovni korak	1.5 m
$\boldsymbol{P_d}$	Pozicija gonilnega stanja	$r_a *$f(N) metrov za čredo
$\boldsymbol{P_c}$	Pozicija zbiralnega stanja	r_a metrov za agentom

Tabela 3.1: Seznam uporabljenih parametrov v simulaciji.

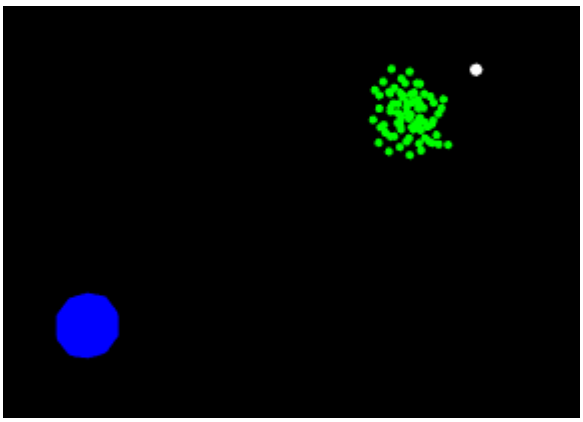
Po nastavitvi parametrov simulacije na parametre s tabele 3.1 zaženemo simulacijo s 74 ovcami ter preverimo delovanje simulacije pri k = 34. Slika 3.6 prikazuje sličico s simulacije, ter prikazuje delovanje ovčarja, ki se trudi potiskati čredo ovc proti cilju, ki je označen z modro barvo. Ovčar je na sliki postavljen za čredo, ter bo zmagal ko bo GCM ovc prišel znotraj modrega kroga.



Slika 3.6: Sličica delovanja simulacije.

3.7 Vizualizacija

Za boljšo vizualizacijo simulacije lahko obarvamo ovce glede na njihovo stanje. Ovce obarvamo glede na njihovo oddaljenost od GCM, kjer ovce znotraj razdalje f(N) obarvamo zeleno, ovce zunaj razdalje f(N) pa obarvamo oranžno. Pri taki vizualizaciji lahko nato opazujemo kdaj se ovce preveč oddaljijo od sredine črede. Za boljše sledenje ovčarjevih akcij ter stanj dodamo dodatne barve. Najdlje oddaljeno ovco v čredi obarvamo rdeče, točko kamor se hoče ovčar premakniti pa označimo tako, da povežemo ovčarja s točko z belo črto. S to vizualizacijo lahko preprosto sledimo menjavam ovčarjevih stanj, saj lahko gledamo ali so vse ovce v čredi, in vidimo kje je gonilna točka. V vizualizaciji prav tako vdimo, kdaj se ovce oddaljijo preveč stran od GCM, ter sledimo najdlje oddaljeni ovci in opazujemo, kje je ovčarjeva zbiralna točka, s katero bo ovco pripeljal nazaj v čredo. Slika 3.7 prikazuje ovčarja v gonilnem stanju. Ovčar na sliki je postavljen na gonilni točki ter potiska čredo ovc proti cilju. Slika 3.8 pa prikazuje ovčarja v zbiralnem stanju. Njegova zbiralna točka P_c se nahaja za najdlje oddaljeno ovco, glede na GCM ovc in je prikazana kot konec bele črte, ki prihaja iz ovčarja.

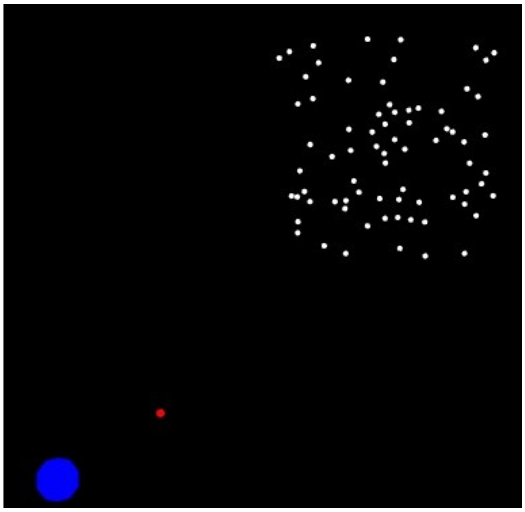


Slika 3.7: Ovčar v gonilnem stanju.

Slika 3.8: Ovčar v zbiralnem stanju.

Poglavje 4 Testiranje ter primerjava rezultatov

Simulacijo s članka so testirali na nekaj načinov, eden od načinov je bil prikaz zmag ter porazov pri čredenju ovc, kjer je ovčar skušal v 8000 korakih pripeljati čredo ovc na cilj. Simulacija je izvedena tako, da naključno postavimo ovce v zgornjo desno četrtino kvadrata velikosti 150m x 150m, ter ovčarja na naključno mesto na desno spodnjo četrtino. Velikost ovc večamo od 1 do 150, ter za testiranje večamo tudi k najbljižjih sosedov od 1 do N-1. Skrajni levi spodnji kot kvadrata je cilj, in če ovčar potrebuje več kot 8000 časovnih korakov ni bil uspešen pri usmerjanju črede ovc. Na sliki 4.1 vidimo začetno stanje simulacije, kjer so bele pike ovce, rdeča pika ovčar ter modra pika cilj.



Slika 4.1: Sličica začetnega stanja iz simulacije.

Prva stvar, ki jo moramo zagotoviti je poenoten sistem za razdalje. Ovcam pripišemo velikost kroga s polmerom štirih pikslov. Krog velikosti ovce vzamemo za merilo. Če predpostavimo, da imamo ovce, ki so dolge približno dva metra v dolžino, lahko določimo skalo velikosti za piksle, kjer štirje piksli dolžine predstavljajo en meter. Vse dolžine v tabeli 3.1 lahko sedaj pretvorimo iz metrov v piksle. Velikost LxL kvadrata sedaj pretvorimo v velikost 600x600 pikslov, če postavimo težišče kvadrata v izhodišče koordinatnega sistema, postavimo ovce v prvi kvadrant, ovčarja pa v tretjega, cilj pa postavimo na točko v koordinati (-300, -300). Program preuredimo tako, da na začetku kjer polnimo seznam ovc poskrbimo, da jih vedno postavimo na prvi kvadrant, ovčarja pa v tretji kvadrant koordinatnega sistema. Prav tako

upravljalcu agentov dodamo parametre, ki bodo pomagali shranjevati rezultate testiranja. Dodamo števce za število izračunanih sličic, ki zadostuje pogoju za neuspešno vodenje ovc, ko število sličic doseže 8000. Dodamo števec za zmage ter izgube ter število iteracij testiranja določenih parametrov. Za vsak parameter N in k je potrebno pognati 50 iteracij za testiranje, zato, da je statistika zmage in poraza bolj zanesljiva. Na koncu vsake izračunane sličice preverimo ali je GCM oddaljen manj kot 7 metrov od cilja, saj če je GCM dovolj blizu cilja to pripišemo kot zmago za ovčarja, ter to prištejemo števcu zmag, če ovčarju v 8000 sličicah ne uspe doseči cilja, to prištejemo števcu izgub. Ko število iteracij doseže 50 pomeni, da smo dovoljkrat preizkusili algoritem za to kombinacijo N in k, ter se pomaknemo na naslednjo kombinacijo. Za sistematično preverjanje vseh kombinacij napišemo algoritem, ki se bo pravilno sprehajal po prostoru možnih parametrov. Šteti začnemo pri N=1 ter k=1, k ne more biti večji od števila ovc, in v primeru, kjer je N enak k, se pomaknemo na naslednji N, ter k prestavimo nazaj na 1. V tem primeru je naslednji korak N=2 ter k=1, tokrat na vsakih 50 iteracij premaknemo na naslednji k, dokler k ni spet enak N. Na ta način se bomo sprehodili po vseh možnih kombinacijah k in N. Če N doseže več kot 150, zaključimo testiranje in tako poustvarimo testiranje uspešnosti simulacije s članka. Na začetku zanke sedaj nastavimo začetno vrednost k in N, ter zaženemo simulacijo. Ker testiranje lahko traja dolgo časa si prav tako sproti shranjujemo rezultate v tekstovno detoteko.

4.1 Pohitritve

Sedaj se pojavi težava s trajanjem testiranja, ter s hitrostjo računskih operacij. Celoten prostor med N in k je velik 11325 primerov kot prikazuje račun (4.1), kjer za vsako kombinacijo izvajamo 50 iteracij kar se pomnoži v 566250 simulacij. Če preučimo graf s članka vidimo da je veliko simulacij neuspešnih za ovčarja, če predpostavimo, da bo povprečna simulacija potrebovala 3000 sličic za to da ovčar pride do cilja (vsaka neuspešna simulacija potrebuje 8000 sličic, uspešne simulacije pa še vedno potrebujejo nekaj časa da ovčar privede vse ovce na cilj.) je končno število sličic, ki jih je treba izračunati ter izrisati približno 1698750000, pri 60 sličic na sekundo bi potrebovali nekprekinjeno testirati približno 327,7 dni, da bi prišli do konca.

$$N \in [1, 150], \qquad k \in [1, N] \tag{4.1}$$

$$\text{št.}primerov = \frac{150 * (150 + 1)}{2} = 11325$$

$$\text{št.}simulacij = \text{št.}primerov * \text{št}\ ponovitev = \ 11325 * 50 = 566250$$

$$\text{št.}sličic = \text{št.}simulacij * povp.dolžina\ simulacije = 566250 * {\sim}3000$$

$$= 1698750000$$

$$trajanje\ testiranja = \frac{\text{št.}sličic}{60\frac{sličic}{s}} = \frac{1698750000}{60} = 28312500s = 471875min$$

$$= 7864{,}6h = 327{,}7dni$$

Da bi lažje razumeli, kako pohitriti testiranje pogledamo graf uspešnosti s članka, ki je prikazan na sliki 4.2. Graf prikazuje uspešnost različnih kombinacij k in N z obarvanim kvadratom na lokaciji v grafu, ki ustreza x in y koordinati, kjer je x število ovc, ter y število najbližjih sosedov, ki jih zazna ovca. Barva kvadrata na koodrinati predstavlja uspešnost testiranja, kjer je bela barva največja uspešnost algoritma in je ovčar v vseh 50 iteracijah uspešno pripeljal vse ovce do cilja, temnejša kot je je modra barva kvadrata slabši so rezultati. Če podrobno pogledamo resolucijo grafa, vidimo, da so pri testiranju spuščali približno vsak drugi N ter vsak drugi k. Če torej začnemo z N=2 ter k=2 in testiramo le za vsak sodi N ter vsak sodi k, se nam število primerov kombinacij zmanjša približno za četrtino. S preskakovanjem primerov nam ostane 2832 primerov kombinacij oziroma 424800000 sličic, tako se nam časovna zahtevnost za testiranje zniža le na 82 dni, kot prikazuje enačba (4.2).

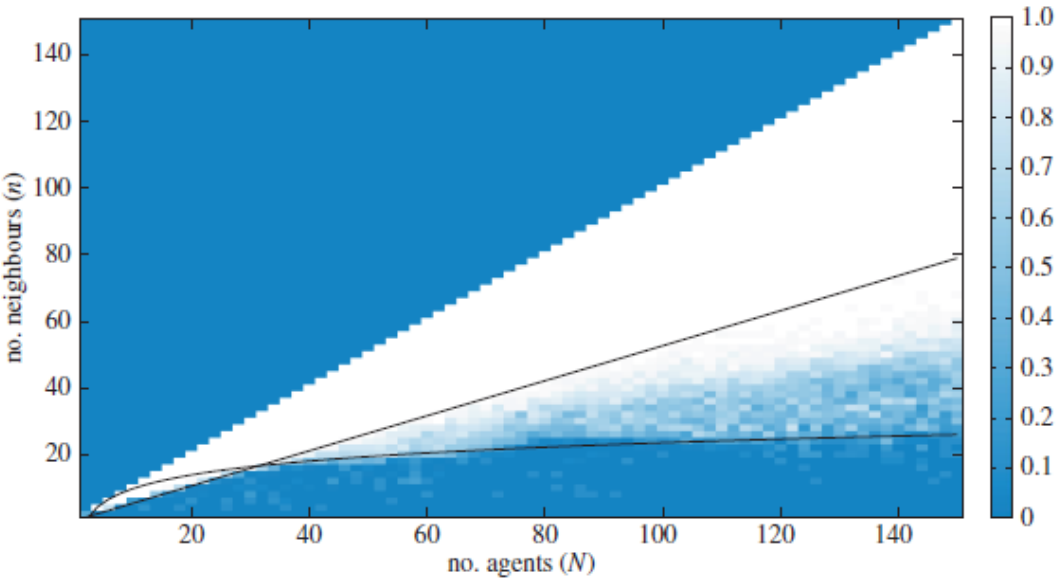
$$\text{št.}primerov = \frac{11325}{4} = 2831{,}25 \cong 2832 \tag{4.2}$$

$$\text{št.}simulacij = 2832 * 50 = 141600$$

$$\text{št.}sličic = 141600 * {\sim}3000 = 424800000$$

$$trajanje\ testiranja = \frac{424800000\ sličic}{60\frac{sličic}{s}} = 7080000s = 118000min$$

$$\cong 1966{,}7h \cong 82dni$$



Slika 4.2: Graf rezultatov s članka.

Naslednja stvar, ki je nepotrebna pri testiranju je izrisovanje sličic, ter omejitev 60 sličic na sekundo, tako simuliranje pretvorimo v matematični problem. V kodi zakomentiramo izrisovanje ter izklučimo omejitev 60 sličic na sekundo, sedaj nam aplikacija namesto na 60 sličicah na sekundo lahko teče tudi na približno 6000 sličic na sekundo pri majhnih številih N ter na približno 1000 sličic na sekundo pri visokih številih N, kar skrajša čas testiranja iz 82 dni na nekaj dni, kot lahko vidimo v enačbi (4.3).

$$trajanje\ testiranja_{maxFPS} = \frac{424800000}{6000} = 70800s = 1180min \cong 20h \tag{4.3}$$

$$trajanje\ testiranja_{minFPS} = \frac{424800000}{1000} = 424800s = 7080min = 118h$$

$$\cong 5dni$$

Postopek smo tudi pohitrili z optimizacijo kode tako, da pri računanju razdalj med agenti ne računamo celotne evklidske razdalje, ki vsebuje operacijo korenjenja. Pri preverjanju razdalj rabimo primerjati polmer območja odbijanja z razdaljo med agenti, vendar rezultat primerjanja ostane enak če primerjamo kvadrata obeh razdalj. Ob primerjanju kvadratov razdalj se izognemo n^2 klicov funkcije korenjenja, ki je pa zelo počasna operacija [7]. Namesto, da to operacijo opravimo med vsako ovco posebej, lahko uporabimo tudi vgnezdeno zanko.

$$O_{dvojna\ for\ zanka}(n^2) \tag{4.4}$$

$$O_{vgnezdena\ for\ zanka}\left(\frac{n*(n+1)}{2}\right)$$

$$n^2 - \frac{n*(n+1)}{2} = 22500 - 11325 = 11175$$

Namesto, da vsakemu agentu posebej prištevamo vektor hitrosti, lahko prvemu agentu prištejemo in isti vektor odštejemo drugemu agentu. S to dodatno se izognemo 11175 primerjav, kot prikazuje enačba (4.4), kar pomeni, da smo se izognili skoraj polovici primerjav. Primer vgnezdene zanke vidimo na sliki 4.3.

```
for( vector<Particle>::iterator p = mParticles.begin(); p != mParticles.end(); ++p ){
    Vec3f currentPos = p->mPos;
    vector<Particle>::iterator p1 = p;
    for( ++p1; p1 != mParticles.end(); ++p1 ) {
        Vec3f nextPos = p1->mPos;
        Vec3f dirToCurrent = currentPos - nextPos;
        float distToCurrent = dirToCurrent.lengthSquared();
        if ( distToCurrent <  zoneRadiusSheepSqrd ) {
            float F = mRepelBetweenSheep;
            dirToCurrent.normalize();
            dirToCurrent *= F;
            p->mAcc += dirToCurrent;
            p1->mAcc -= dirToCurrent;
        }
    }
}
```

Slika 4.3: Primer vgnezdene zanke.

Na začetku programiranja smo si izbrali list kot tip seznama, vendar po preučevanju literature opazimo, da obstaja tudi vektor, ki je v nekaterih primerih boljši od lista. List je vrsta seznama, ki deluje hitrejše od vektorja pri operacijah vstavljanja ali brisanja elementov iz sredine seznama, vendar počasnejši pri dostopanju do elementov. Pri vektorju je treba paziti, da elemente dodajamo na konec seznama, saj bi v primeru dodajanja na začetek seznama morali vsak element prestaviti za eno mesto naprej. Pri simulacijah ovc ne rabimo odstranjevati elementov iz seznama, dodajali pa jih bomo le na začetku, prav tako je bolj pomemben hiter dostop do elementov, saj bomo do njih zelo pogosto dostopali pred izrisom vsake sličice.

Prav tako optimiziramo kodo, tako da preverimo, kje je koda najpočasnejša s pomočjo orodja analize zmogljivosti, ki je vgrajena v Visual Studio. Ko zaženemo program vidimo, da je operacija iskanja k-najbližjih sosedov najbolj zahtevna. Pri sortiranju seznama ovc uporabljamo quicksort funkcijo, ki nam razvrsti vse elemente po velikosti. Za računanje LCM agentov ne potrebujemo seznama urejenega po velikosti, potrebujemo le vedeti kateri agenti so najbližje agentu, ki ga gledamo. Za računanje LCM bi nam zadostovalo, da imamo seznam, kjer je na k-tem mestu v seznamu agent, ki je tudi k-ti po vrsti v urejenem seznamu, ter na eni strani od njega, vsi agenti manjši od k-tega agenta, na desni pa vsi večji po oddaljenosti. Za to zamenjamo operacijo `quicksort()` z `nthelement()` , ki nam sortira seznam na način kot ga rabimo in zniža časovno zahtevnost sortiranja iz O(n*logn) na O(n). Pri preučevanju grafa s članka ter rezultatov ugotovljenih iz njihovih poizkusov, ter s preučevanjem delovanja simulacije prav tako ugotovimo da pri vrednostih k, ki so večje od N/2, začnemo zanesljivo dobivati zmage. Zaradi tega lahko dodatno znižamo število testnih primerov tako, da testiranje za določen N ustavimo ter nadaljujemo na naslednjega, če je k enak N/2 ter imamo vsaj tri ali štiri zaporedne popolne zmage pri vseh pedesetih iteracijah poizkusa, saj se z večanjem k zanesljivost zmag le še povečuje. Namesto 2831 kombinacij k in N se sedaj lahko testiranje zaključi že z 1600 primeri, kar še dodatno zmanjša čas testiranja. Sedaj zaženemo teste, ki delujejo veliko hitreje in jih lahko opravimo v približno 19 urah.

4.2 Rezultati testiranja

S sledenjem navodilom iz članka smo poustvarili delovanje simulacije, ki jo članek opisuje. Da preverimo ali smo pravilno poustvarili simulacijo moramo preveriti ali je uspešnost naše kode podobna uspešnosti kode iz članka. Za ta namen postavimo velikost območja vpliva ovčarja na enako vrednost, kot so jo nastavili za merjenje uspešnosti simulacije, ter zaženemo teste. Ko so opravljeni, dobimo tekstovno datoteko, ki jo lahko primerjamo z rezultati, ki so prikazani na sliki 4.2. Da bi to naredili ročno bi potrebovali zelo veliko časa, zato raje ustvarimo tudi program, ki bo lahko generiral podoben graf kot je na sliki, s podatki, ki smo jih dobili od testiranja.

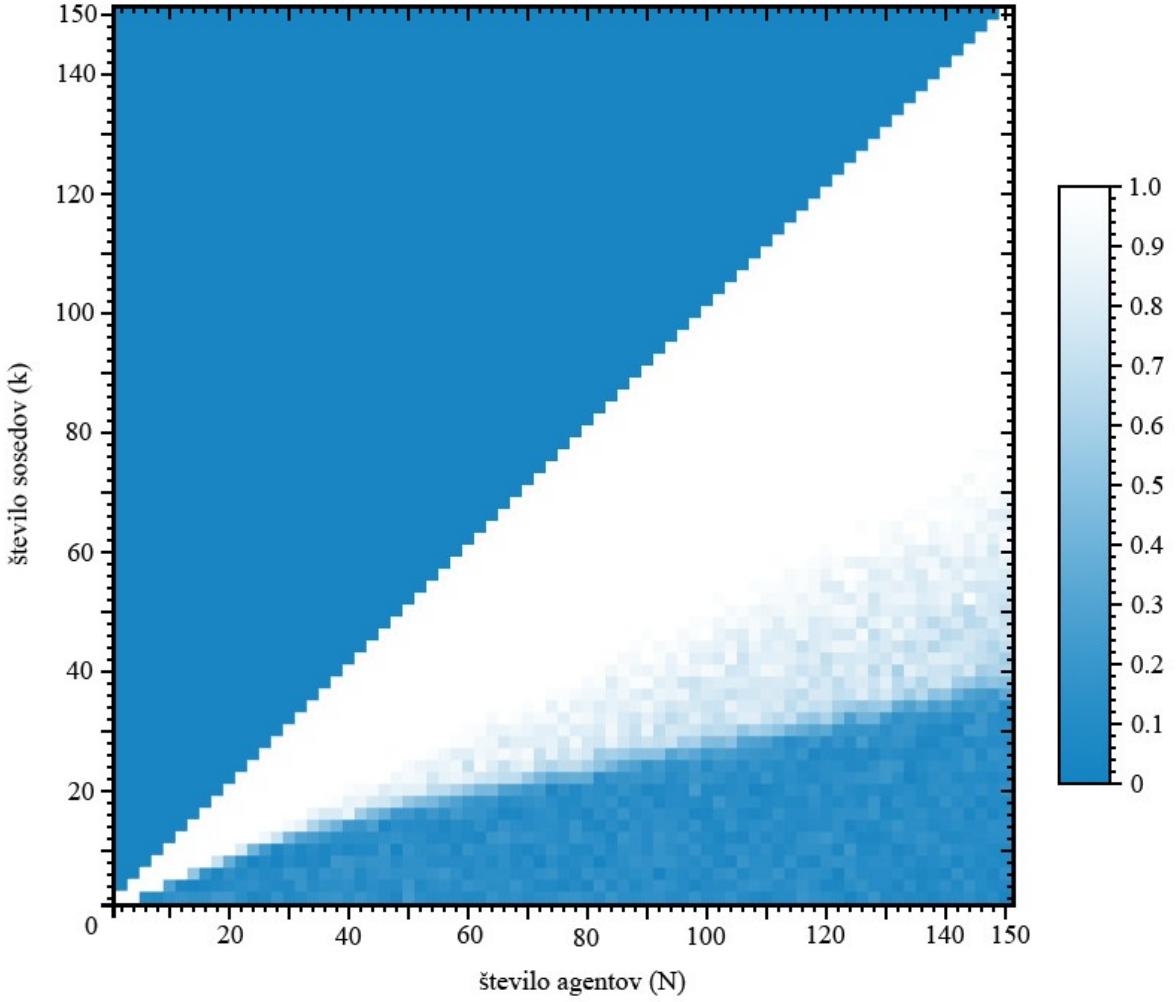
4.2.1 Generiranje grafov

V Visual Studiu ustvarimo nov cinder projekt, s katerim bomo generirali grafe iz podatkov, ki smo jih dobili pri testiranju. Graf je dvodimenzionalno polje s številom ovc za x os, ter k-najbližjih sosedov za y os. Na vsaki koordinati je obarvan kvadrat, kjer vrednost modre barve predstavlja uspešnost simulacije pri tej kombinaciji N ter k. Za generanje grafa se odločimo, da bomo s pomočjo OpenGL grafičnih sposobnosti ustvarili polje, zgrajeno iz kvadratov, ter

jih pobarvali glede na uspešnost simulacije, ki jo preberemo iz tekstovne datoteke. V inicializaciji ustvarimo dva vektorja, ki bosta hranila x in y koordinate, ter vektor, ki bo hranil število zmag. Za graf se odločimo, da bo sestavljen iz kvadratov velikih 8x8 pikslov, kar pomeni, da pri resoluciji kombinacij, ki smo jih uporabili, rabimo okno, ki je veliko vsaj 600x600 pikslov, da vidimo cel graf. V `update()` funkciji najprej odpremo tekstovni dokument, ter preberemo vse podatke v vektorje. V `draw()` funkciji se nato sprehodimo po tabeli ter izrisujemo kvadrate na koordinatah k in n, za barvo kvadrata nastavimo barvo, ki je odvisna od zmag. Brava, ki jo določimo rdečemu, zelenemu ali modremu kanalu za RGB vrednost je lahko med 0 in 1. Da dobimo podobno skalo, kot je v grafu s članka, nastavimo vrednost RGB na (25, 133, 195) ter vse vrednosti enakomerno večamo do (255, 255, 255) v odvisnosti od števila zmag, saj tako premikamo modro barvo proti beli. Zaradi odstranjevanja testnih primerov za pohitritev testiranja, rabimo sedaj tudi dopolniti manjkajoče kvadrate v grafu. Ko pri sprehajanju po vektorju pridemo do primera kjer k pade nazaj na začetno vrednost, nadaljujemo z generiranjem belih kvadratov dokler ne pridemo do vrednosti k, ki je enaka n, kjer nadaljujemo z modro barvo ter tako dopolnemo graf. V Photoshopu nato uredimo grafe tako, da jim dodamo skalo ter naredimo grafe bolj pregledne.

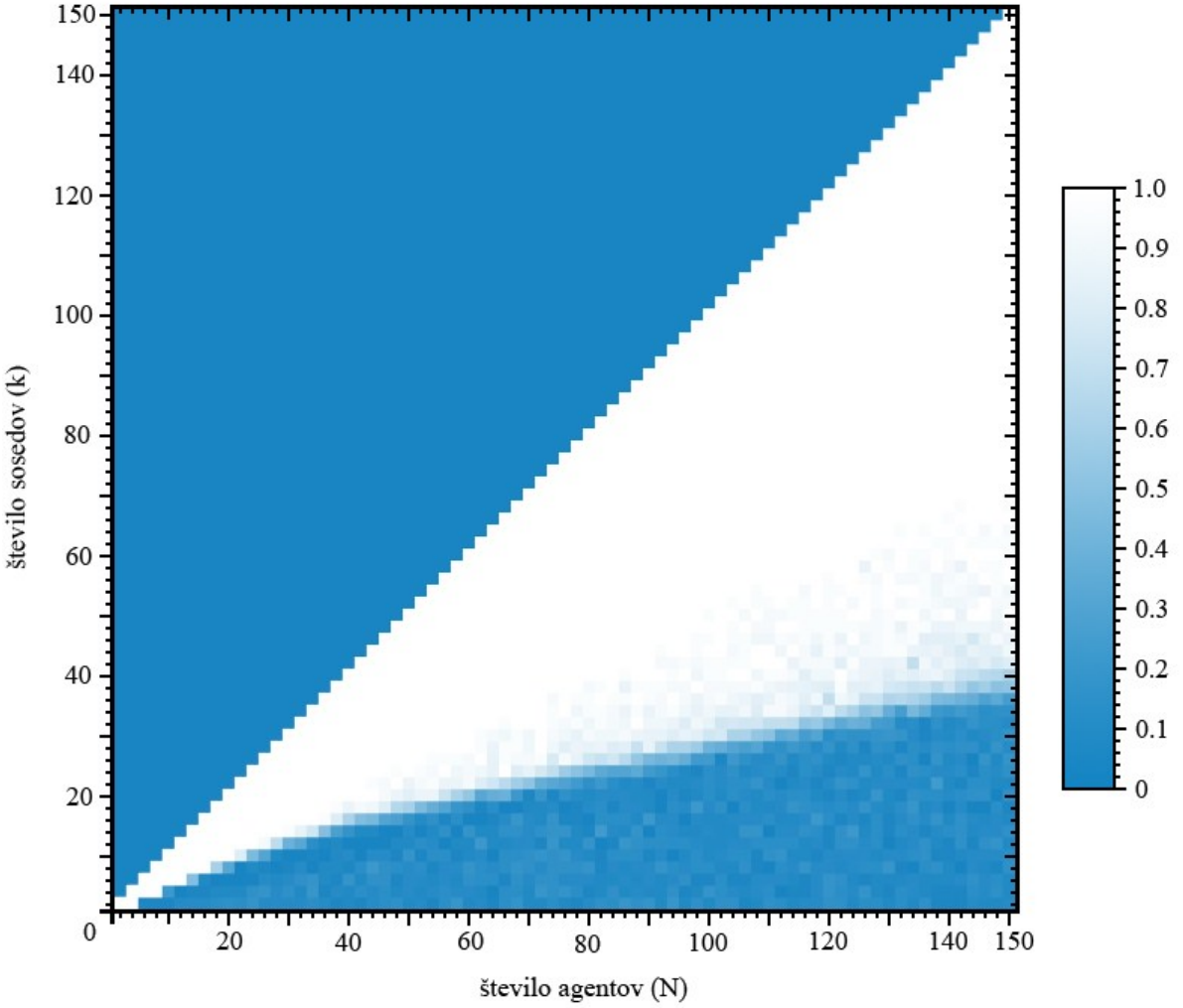
4.3 Analiza grafov rezultatov

Vidimo, da so rezultati grafa na sliki 4.4 podobni rezultatom s članka. Opazimo podobno krivuljo kjer začne ovčar bolj pogosto zmagovati, prav tako so vsi rezultati pri k večjem od n/2 vedno uspešni. Čeprav je meja kjer začne ovčar zmagovati pri višjih N dvignjena ter ne spada več pod krivuljo $3*\log_2(N)$, kot je opisano v članku, je graf še vedno podoben grafu iz članka, namreč pri njihovem grafu prav tako pri visokem številu ovc opazimo točke izven krivulje kjer ovčar ne zmaga.



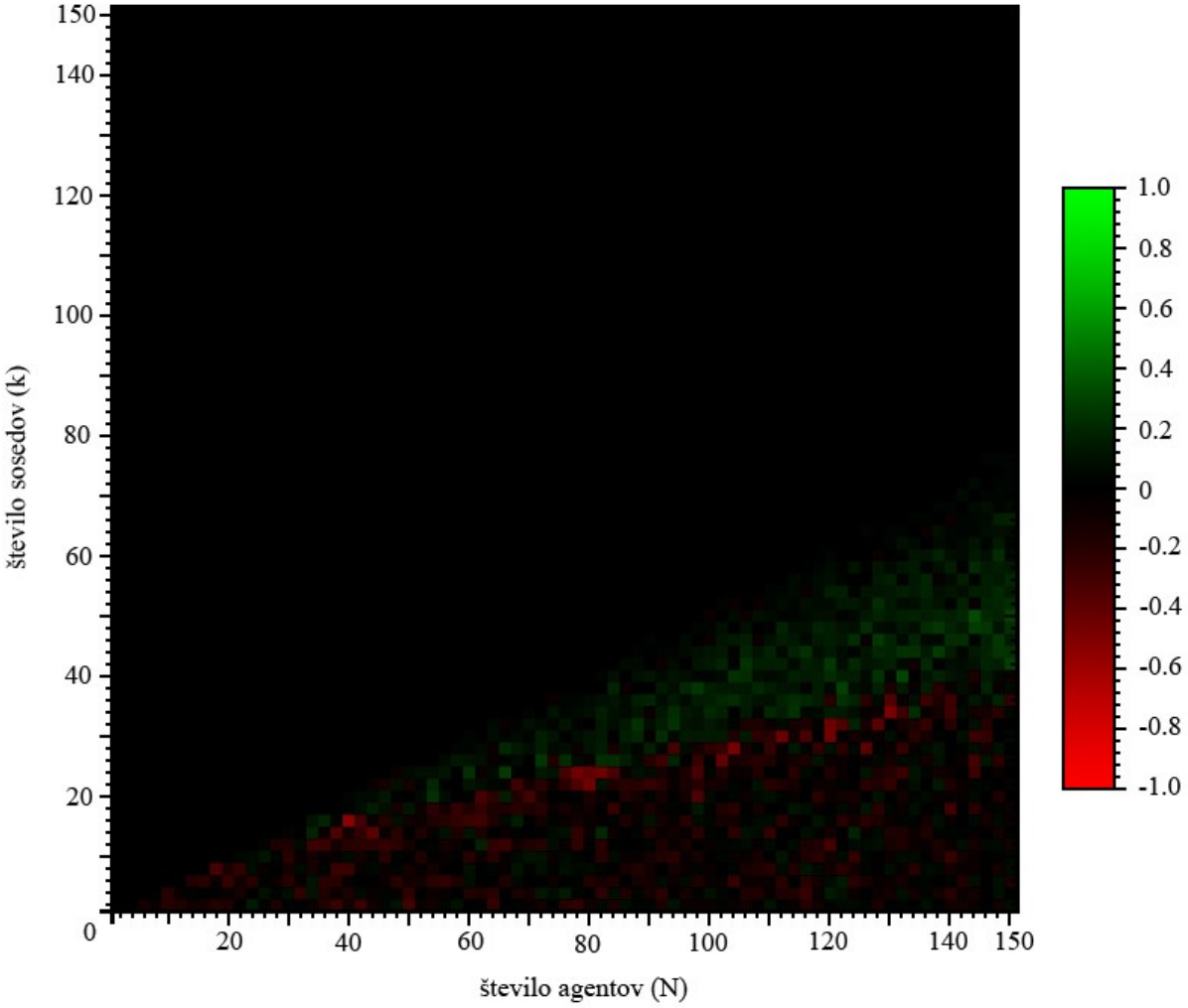
Slika 4.4: Rezultati testiranja kode po parametrih v članku.

Odločimo se povečati r_s na takega kot je v tabeli, nakar ponovimo teste. Zanima nas zakaj so ravno r_s pomanjšali za prikaz grafa zmag. Po opravljenih testih ugotovimo, da se pri povečani vrednosti r_s spodnja krivulja ne spremeni vendar se pa precej izboljšajo rezultati nad krivuljo. To je bil verjetno namen znižanja razdalje zaznave ovčarja, saj je tako graf bolj berljiv, ter so rezultati z njega bolje razvidni. Na sliki 4.5 je prikazan graf rezultatov simulacije s parametri nastavljenemi po tabeli 3.1.



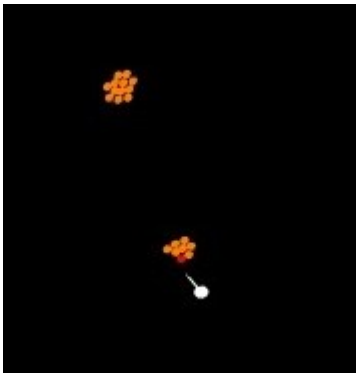
Slika 4.5: Rezultati testiranja kode po parametrih v tabeli.

Na sliki 4.6 vidimo z zeleno označene točke kjer se je rezultat izboljšal, prav tako vidimo točke obarvane rdeče kjer so se rezultati poslabšali. Večino poslabšanih rezultatov je pod krivuljo, kjer so je število zmag nizko, izboljšajo se pa primeri, kjer ovčar začne zmagovati. S povečano razdaljo vpliva ovčar lažje pomaga ovcam priti do cilja v primerih, kjer je prej imel težave.



Slika 4.6: Razlika med grafoma r_s.

Pri opazovanju delovanja simulacije pri nizkih številih ovc ter številu sosedov blizu polovice števila ovc pogosto pridemo do primera, kjer se ovce razpolovijo v dve skupini ter jih ovčar ne more več združiti v eno samo čredo. Do tega pride zaradi razdalje vpliva ovčarja v primerjavi z velikostjo čred ovc. Ovčar pride preveč blizu skupini ovc, ki se zaradi premajhnega števila sosedov razpolovi. Ovčar se iz gonilnega stanja prestavi v zbiralno stanje, ter se premakne za najdlje oddaljeno ovco ter jo začne potiskati nazaj proti GCM. Ker je druga skupina ovc še vedno znotraj vpliva ovčarja ter so sile odbijanja konstantne glede na oddaljenost, se druga skupina ovc začne oddaljevati stran od GCM z enako hitrostjo kot ovčar potiska prvo skupino proti GCM. Pridemo do mrtve točke v programu, kot vidimo na sliki 4.7, v katerem bo ovčar potiskal ovce vse dokler ne bo poteklo število korakov za neuspešno opravljen poizkus.



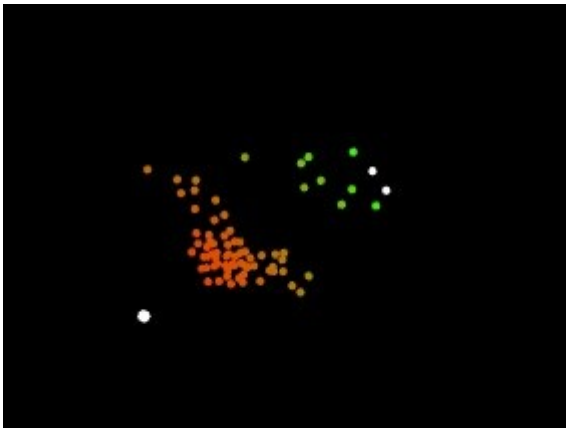
Slika 4.7: Mrtva točka v simulaciji pri majhnem N in k blizu N/2.

4.4 Implementirane izboljšave

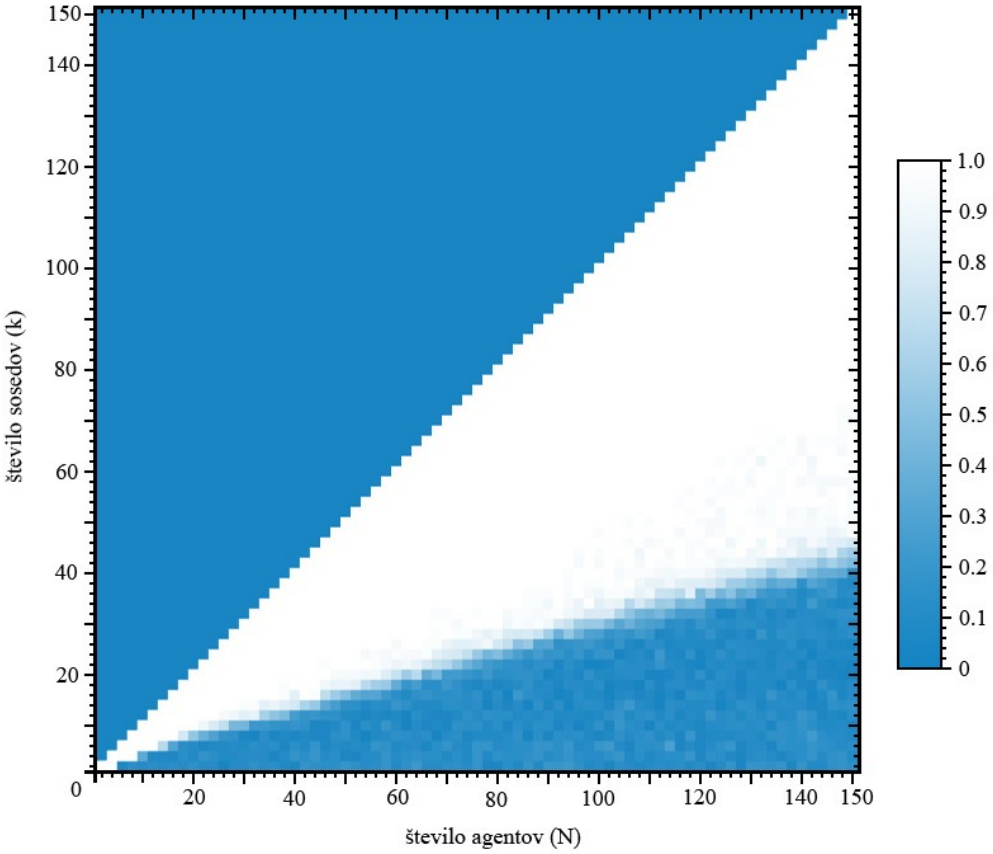
V članku je sila odbijanja od ovčarja konstantna, kar pomeni, da ovce bežijo stran od ovčarja z enako hitrostjo na najdaljši oddaljenosti, kot ovce od katerih je ovčar oddaljen le nekaj metrov. Tako obnašanje se zdi nenaravno, zato nas zanima, kaj se zgodi, če spremenimo simulacijo tako, da spremenimo odbijanje od ovčarja na linearno enačbo, kjer ovce, ki so najbližje ovčarju odbijamo z največjo silo, vse ovce naprej v čredi pa z vedno nižjo silo, tako pridemo do vpliva kjer se ovce, ki so na drugi strani črede kot ovčar, manj zavedajo nevarnosti. V tem primeru se vodenje črede zanaša bolj na to, da ovce najbližje ovčarju potiskajo ovce pred njimi v smeri proti cilju. S tem se teoretično tudi znebimo mrtve točke s slike 4.8, saj sedaj sila od ovčarja ni več konstantna, ampak je odvisna od oddaljenosti agentov. Skupina bližje ovčarju se bo približevala GCM hitreje kot se skupina, ki beži od ovčarja, tako lahko združimo čredi ovc skupaj ter omogočimo ovčarju, da se premakne nazaj v gonilno stanje. Simulacijo popravimo tako, da spremenimo silo ovčarja iz konstantne v linearno, kot je prikazano v enačbi (4.5), kjer je F izračunana sila, r_d razdalja ovce od ovčarja, ostali vrednosti pa sta opisani v tabeli 3.1.

$$F = \left(1 - \frac{r_d}{r_s}\right) * \rho_s \tag{4.5}$$

Linearno nižanje sile z razdaljo prikazuje slika 4.8, kjer je ovčar velika bela točka levo spodaj. Največjo silo odbijanja od ovčarja čutijo ovce, ki so najbložje ovčarju, ter so obarvane rdeče, dlje, kot so oddaljene ovce, manj sile čutijo. Najdlje oddaljene ovce znotraj območja vpliva ovčarja so obarvane zeleno, ovce zunaj vpliva ovčarja pa so obarvane belo.



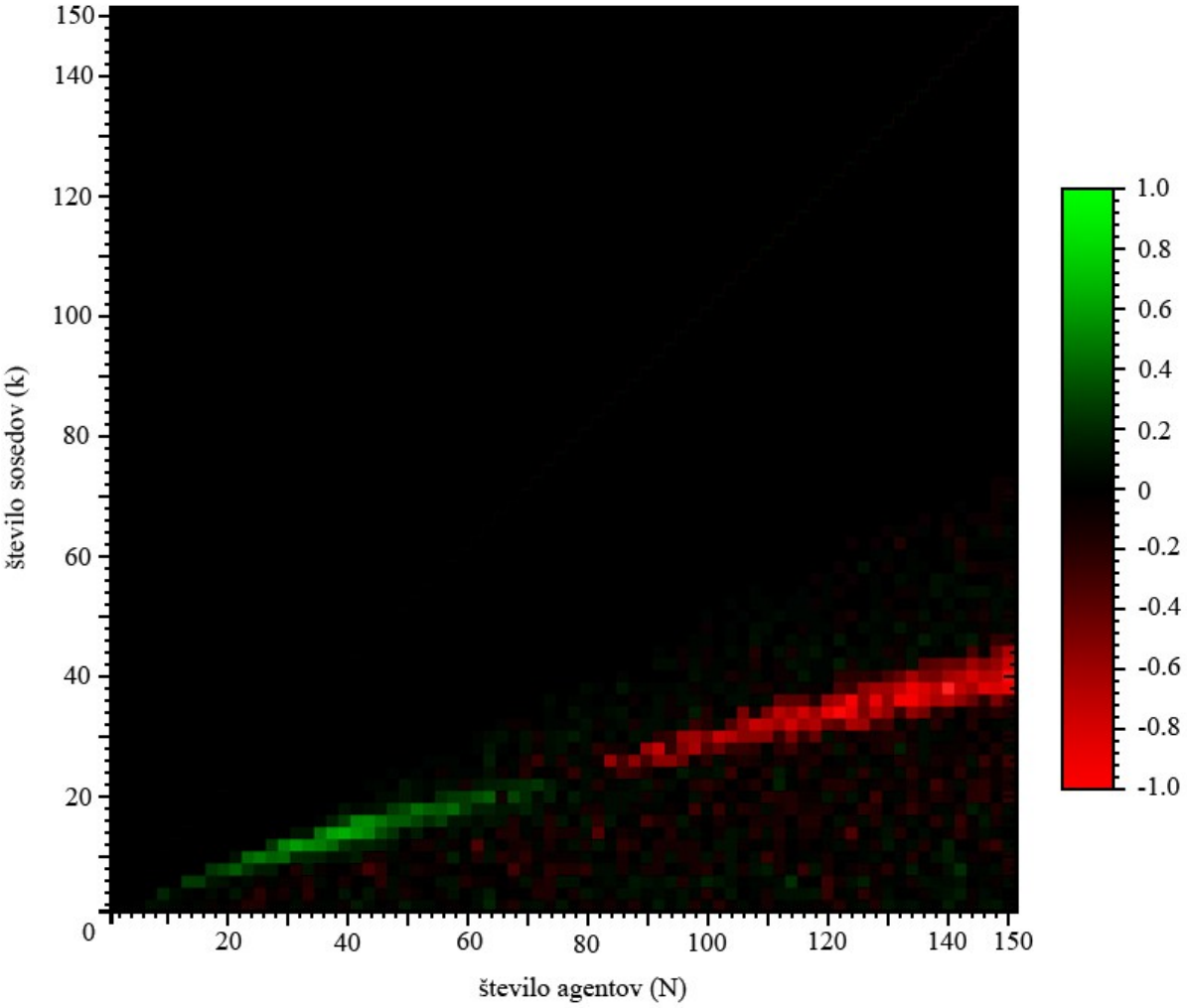
Slika 4.8: Prikaz moči linearne sile odbijanja ovčarja.



Slika 4.9: Graf s spremenjeno silo odbijanja od ovčarja.

Na sliki 4.9 vidimo rezultate testiranja s spremenjenim odbijanjem ovc od ovčarja. Na sliki 4.10 so tako kot za prejšni graf sprememb, prikazane razlike med grafom z rezultati spremenjeno silo odbijanja ter grafom z rezultati s parametri iz tabele, ostali parametri za spremenjeno verzijo simulacije so enaki kot v tabeli 3.1. Opazimo, da se program izboljša pri

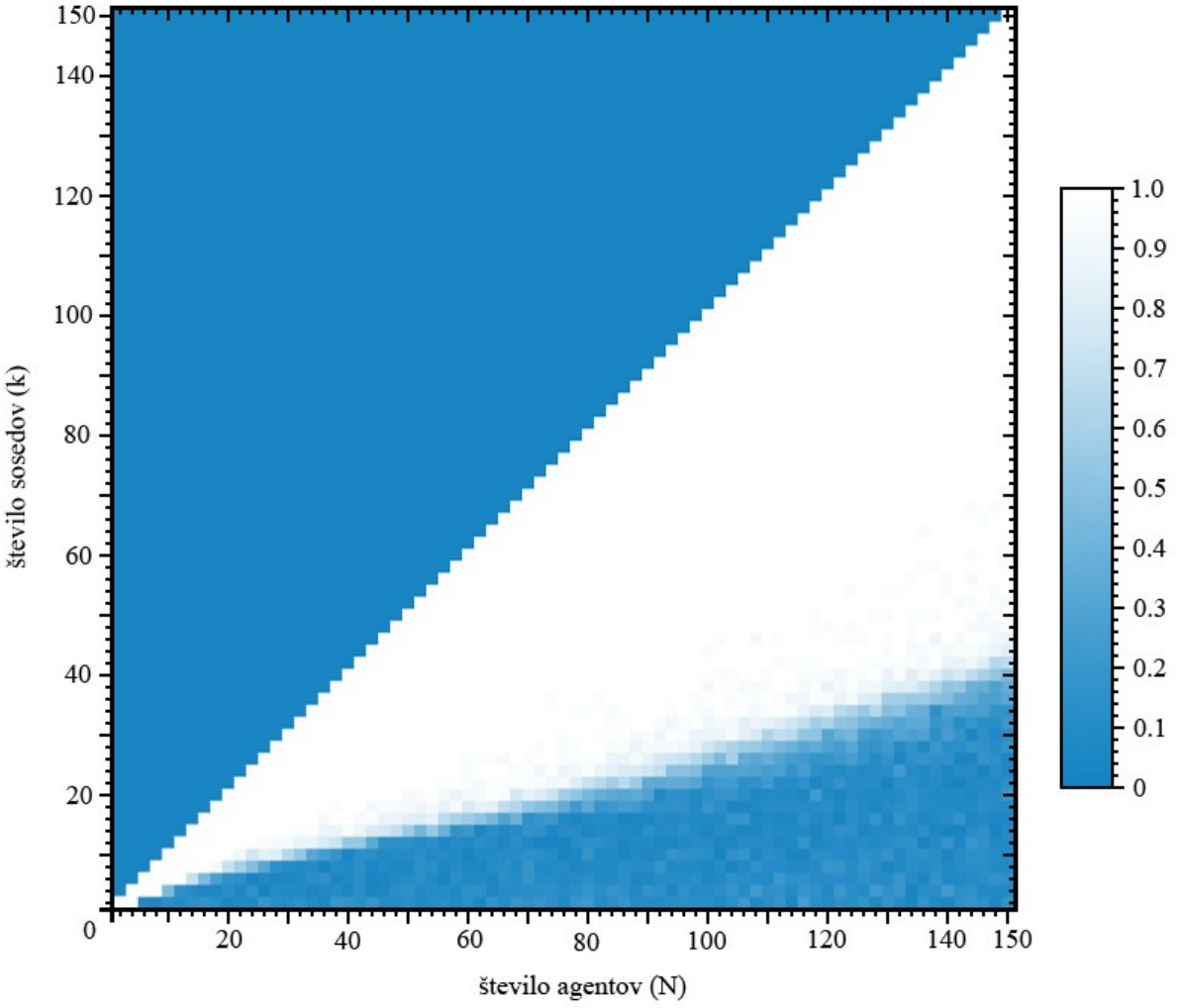
številu ovc manjšem od 80, vendar se pri večjem številu ovc začne število zmag manjšati v primerjavi z rezultati brez spremenjene sile ovčarja. Pri majhnih številih ovc je sprememba služila kot izboljšanje



Slika 4.10: Graf razlike med r_s = 65m ter Izboljšano verzijo.

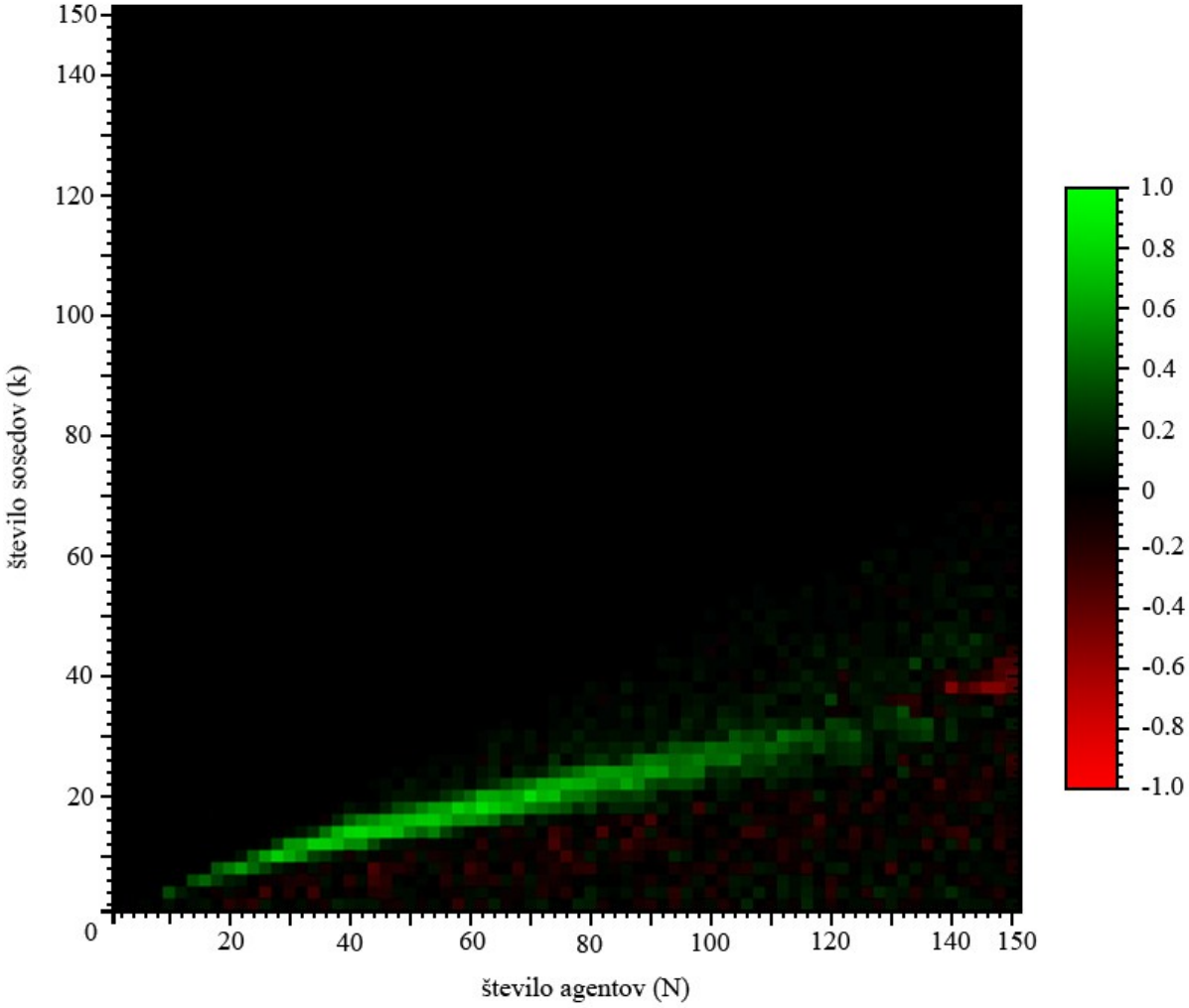
Pri večjih številih ovc se pojavi druga težava, saj ovce uidejo iz območja vpliva ovčarja ter se razpršijo, ko na njih ne deluje več čredni nagon. Če se ovčar znajde med obema čredama bo algoritem za zbiralno stanje onemogočil ovčarja, ker bo zaradi kombinacije črednega nagona in prostega pasenja ovc najdlje oddaljena ovca enkrat na eni čredi, drugič na drugi. Ker bosta dve čredi, ki se prosto paseta, bo najdlje oddaljena ovca na eni od teh čred, vendar takoj ko se ovčar približa tej čredi se ovcam vključi čredni nagon, te ovce se nato združijo v čredo, zaradi česar postane nova najdlje oddaljena ovca tista, ki je od GCM najdlje oddaljena v drugi čredi. Ovčar se nato približa drugi čredi in postopek se ponovi.

Zanima nas, če je zaradi tega za novo metodo velikost vpliva ovčarja premajhna. V članku zasledimo, da so za testiranje modela zvišali r_s tudi do 75 metrov, zato poskusimo za spremenjeno silo povečati tudi vpliv ovčarja ter preveriti, če se zaradi tega rezultati izboljšajo.



Slika 4.11: Graf uspešnosti linearnega odbijanja pri $r_s = 75$.

Za graf na sliki 4.11 smo povečali tudi silo odbijanja ovčarja, saj ovčar nikoli ne bo bližje ovcam kot $3r_a$. Silo ρ_s spremenimo iz 1 na 1.12, saj zaradi linearnega padanja sile z razdaljo od ovčarja na agente vpliva manjša sila na razdalji $3r_a$ kot sila ρ_s napisana v tabeli. S pomočjo grafa na sliki 4.12, kjer primerjamo graf s parametri iz tabele ter graf s povečano razdaljo območja vpliva ovčarja z linearno silo odbijanja, ugotovimo, da smo zelo izboljšali rezultate od simulacije s povečavo območja vpliva r_s za samo 15 metrov. Pri večini primerov števil ovc smo zmanjšali število k najbližjih sosedov potrebnih za uspeh ovčarja tudi za 10, ter dosegli boljše rezultate tudi nad mejo, kjer začne ovčar zmagovati.



Slika 4.12: graf razlike med rezultati s tabele in lineranim odbijanjem.

Na sliki 4.12 vidimo z zeleno obarvano območje kjer so izboljšani rezultati. Na grafu se vidi močno obarvano zeleno polje, kjer se meja najmanjšega števila k-najbližjih sosedov zniža. Sprememba sile iz konstantne v linearno postane veliko izboljšanje pri povečavi območja ovčarja, saj rabimo le poskrbeti, da v primeru razdelitve črede uspe ovčar priti za najdlje oddaljeno ovco, ter še vedno obdrži ostale ovce znotraj svojega območja. Tako mu lahko vedno uspe združiti ločeni čredi nazaj v celoto, ter jo spet peljati proti cilju.

Poglavje 5 Sklepne ugotovitve

V diplomski nalogi smo poustvarili simulacijo čredenja ovc s članka [1]. To simulacijo smo nato preučili in testirali ter dobili podobne rezultate uspešnosti kot pri simulaciji v članku. Poleg testiranja simulacije smo tudi preučili obnašanje ovc ter poskušali najti v kakšnih primerih ovčar ne uspe privesti črede do cilja in zakaj v simulaciji do tega pride. Ko imamo veliko število ovc, ki jih poizkušamo v eni čredi pripeljati do cilja, je uspešnost zelo odvisna od števila najbljižjih sosedov, ki jih ovce zaznajo. Pri manjših številih najbljižjih sosedov se ovce nočejo združevati v enotno čredo, ter se raje zgostijo v manjše skupine, ki jih nato ovčar poizkuša imeti združene v čredi. Pri takih primerih, kjer se ovce ne obnašajo naravno je najbolj primerno nastaviti ovčarju obnašanje, kjer ne poskuša pripeljati cele črede na cilj, vendar jih raje razdrobiti na manjše črede, ki so bolj primerne za določeno vrednost najbližjih sosedov, ter vsako manjšo čredo posebej pripeljati na cilj. Ovčarski psi lahko sami čredijo več kot sto ovc naenkrat, zato je pristop razdrobitve ovc na manjše črede usmerjeno na boljše rezultate pri simulaciji, kot pri poustvarjanju naravnega vedenja psov ovčarjev. Kar bi bilo zanimivo preučiti je vpliv večih ovčarskih psov na čredo, saj z linearno silo odbijanja od ovčarja lahko čredo vodimo proti cilju tako, da ovčarji z večih strani potiskajo čredo proti željeni smeri, ter lahko s približevanjem in oddaljevanjem od črede lažje oblikujejo obliko ter smer v katero se čreda premika. Prav tako bi bilo lažje ujeti ovce, ki se oddaljijo od črede, saj lahko le en ali dva ovčarja opravita to nalogo, med tem ko ostali ovčarji še vedno potiskajo čredo proti cilju.

Literatura

[1] Strömbom, D., Mann, R. P., Wilson, A. M., Hailes, S., Morton, A.J., Stumpter, D. J. T. & King, A. J., 2014, "Solving the shepherding problem: heuristics for herding autonomous, interacting agents", Journal of The Royal Society Interface, 11(100), 20140719.

[2] Shepherd. *Wikipedia* [Online] Dosegljivo: http://en.wikipedia.org/wiki/Shepherd [Dostopano: Marec 2015]

[3] Lien, J.-M., Bayazit, B., Sowell, R.T., Rodriguez, S., Amato, N.M., 2004, "Shepherding behaviors", In: Proceedings of ICRA'04, IEEE International Conference on Robotics and Automation, Vol. 4, pp. 4159-4164.

[4] Reynolds, C., Boids [Online] Dosegljivo: http://www.red3d.com/cwr/boids/ [Dostopano: Marec 2015]

[5] The Barbarian Group, About Cinder [Online] Dosegljivo: http://libcinder.org/about/ [Dostopano: Marec 2015]

[6] Adobe, What is Photoshop? [Online]Dosegljivo: https://helpx.adobe.com/photoshop/how-to/photoshop-cc.html [Dostopano: Marec 2015]

[7] Methods of computing square roots, *Wikipedia* [Online] Dosegljivo: http://en.wikipedia.org/wiki/Methods_of_computing_square_roots [Dostopano: Marec 2015]

[8] Boids, Wikipedia [Online] Dosegljivo: http://en.wikipedia.org/wiki/Boids [Dostopano: Marec 2015]

[9] Bird migration, *Wikipedia* [Online] Dosegljivo: http://en.wikipedia.org/wiki/Bird_migration#Orientation_and_navigation [Dostopano: Marec 2015]

[10] Schoenian S., Sheep A Beginner's Guide to Raising Sheep Dosegljivo: http://www.sheep101.info/201/behavior.html [Dostopano: Marec 2015]

[11] Vo, C., 2014, "Robust and Reusable Methods for Shepherding and Visibility-Based Pursuit", doktorska disertacija, George Mason University, Fairfax, Virginia.

[12] Microsoft, Application Development [Online] Dosegljivo: http://www.visualstudio.com/explore/application-development-vs [Dostopano: Marec 2015]

[13] Delgado-Mata, C., Ibanez-Martinez, J., Bee, S.; Ruiz-Rodarte, R., 2007, "On the Use of Virtual Animals with Artificial Fear in Virtual Environments", *New Generation Computing,* 25(2): 145-169.

[14] Hartman, C., Beneš, B., 2006, "Autonomous Boids" Computer Animation and Virtual Worlds, 17(3-4), 199-206.